Closest Pair of Points Algorithms

Martin Rau and Tobias Nipkow

March 17, 2025

Abstract

This entry provides two related verified divide-and-conquer algorithms solving the fundamental *Closest Pair of Points* problem in Computational Geometry. Functional correctness and the optimal running time of $\mathcal{O}(n \log n)$ are proved. Executable code is generated which is empirically competitive with handwritten reference implementations.

Contents

1 Common

theory *Common*
imports
HOL−Library.Going-To-Filter
Akra-Bazzi.Akra-Bazzi-Method
Akra-Bazzi.Akra-Bazzi-Approximation
HOL−Library.Code-Target-Numeral
Root-Balanced-Tree.Time-Monad
begin

type-synonym *point* = *int* ∗ *int*

1.1 Auxiliary Functions and Lemmas

1.1.1 Time Monad

lemma *time-distrib-bind*:
time (*bind-tm tm f*) = *time tm* + *time* (*f* (*val tm*))
unfolding *bind-tm-def* **by** (*simp split*: *tm.split*)

lemmas *time-simps* = *time-distrib-bind tick-def*

lemma *bind-tm-cong*[*fundef-cong*]:
assumes ⋀*v*. *v* = *val n* ⟹ *f v* = *g v m* = *n*
shows *bind-tm m f* = *bind-tm n g*
using *assms* **unfolding** *bind-tm-def* **by** (*auto split*: *tm.split*)

1.1.2 Landau Auxiliary

The following lemma expresses a procedure for deriving complexity properties of the form $t \in O[m \text{ going-to at-top within } A](f \circ m)$ where

- t is a (timing) function on same data domain (e.g. lists),
- m is a measure function on that data domain (e.g. length),
- t' is a function on *nat*,
- A is the set of valid inputs for the data domain. One needs to show that
- t is bounded by $t' \circ m$ for valid inputs
- $t' \in O(f)$ to conclude the overall property $t \in O[m \text{ going-to at-top within } A](f \circ m)$.

lemma *bigo-measure-trans*:
fixes *t* :: ′*a* ⇒ *real* **and** *t*′ :: *nat* ⇒ *real* **and** *m* :: ′*a* ⇒ *nat* **and** *f* ::*nat* ⇒ *real*
assumes ⋀*x*. *x* ∈ *A* ⟹ *t x* ≤ (*t*′ *o m*) *x*

```
    and t' ∈ O(f)
    and ⋀x. x ∈ A ⟹ 0 ≤ t x
  shows t ∈ O[m going-to at-top within A](f o m)
proof −
  have 0: ⋀x. x ∈ A ⟹ 0 ≤ (t' o m) x by (meson assms(1,3) order-trans)
  have 1: t ∈ O[m going-to at-top within A](t' o m)
    apply(rule bigoI[where c=1]) using assms 0
    by (simp add: eventually-inf-principal going-to-within-def)
  have 2: t' o m ∈ O[m going-to at-top](f o m)
    unfolding o-def going-to-def
    by(rule landau-o.big.filtercomap[OF assms(2)])
  have 3: t' o m ∈ O[m going-to at-top within A](f o m)
     using landau-o.big.filter-mono[OF -2] going-to-mono[OF -subset-UNIV] by
blast
  show ?thesis by(rule landau-o.big-trans[OF 1 3])
qed

lemma const-1-bigo-n-ln-n:
  (λ(n::nat). 1) ∈ O(λn. n * ln n)
proof −
  have ∃N. ∀(n::nat) ≥ N. (λx. 1 ≤ x * ln x) n
  proof −
    have ∀(n::nat) ≥ 3. (λx. 1 ≤ x * ln x) n
    proof standard
      fix n
      show 3 ≤ n ⟶ 1 ≤ real n * ln (real n)
      proof standard
        assume 3 ≤ n
        hence A: 1 ≤ real n
          by simp
        have B: 1 ≤ ln (real n)
          using ln-ln-nonneg' ‹3 ≤ n› by simp
        show 1 ≤ real n * ln (real n)
          using mult-mono [OF A B] by simp
      qed
    qed
    thus ?thesis
      by blast
  qed
  thus ?thesis
    by auto
qed
```

1.1.3 Miscellaneous Lemmas

```
lemma set-take-drop-i-le-j:
  i ≤ j ⟹ set xs = set (take j xs) ∪ set (drop i xs)
proof (induction xs arbitrary: i j)
  case (Cons x xs)
```

```
  show ?case
  proof (cases i = 0)
    case True
    thus ?thesis
      using set-take-subset by force
  next
    case False
    hence set xs = set (take (j − 1) xs) ∪ set (drop (i − 1) xs)
      by (simp add: Cons diff-le-mono)
    moreover have set (take j (x # xs)) = insert x (set (take (j − 1) xs))
      using False Cons.prems by (auto simp: take-Cons')
    moreover have set (drop i (x # xs)) = set (drop (i − 1) xs)
      using False Cons.prems by (auto simp: drop-Cons')
    ultimately show ?thesis
      by auto
  qed
qed simp

lemma set-take-drop:
  set xs = set (take n xs) ∪ set (drop n xs)
  using set-take-drop-i-le-j by fast

lemma sorted-wrt-take-drop:
  sorted-wrt f xs ⟹ ∀ x ∈ set (take n xs). ∀ y ∈ set (drop n xs). f x y
  using sorted-wrt-append[of f take n xs drop n xs] by simp

lemma sorted-wrt-hd-less:
  assumes sorted-wrt f xs ⋀x. f x x
  shows ∀ x ∈ set xs. f (hd xs) x
  using assms by (cases xs) auto

lemma sorted-wrt-hd-less-take:
  assumes sorted-wrt f (x # xs) ⋀x. f x x
  shows ∀ y ∈ set (take n (x # xs)). f x y
  using assms sorted-wrt-hd-less [of f ‹x # xs›] in-set-takeD [of - n ‹x # xs›]
  by auto

lemma sorted-wrt-take-less-hd-drop:
  assumes sorted-wrt f xs n < length xs
  shows ∀ x ∈ set (take n xs). f x (hd (drop n xs))
  using sorted-wrt-take-drop assms by fastforce

lemma sorted-wrt-hd-drop-less-drop:
  assumes sorted-wrt f xs ⋀x. f x x
  shows ∀ x ∈ set (drop n xs). f (hd (drop n xs)) x
  using assms sorted-wrt-drop sorted-wrt-hd-less by blast

lemma length-filter-P-impl-Q:
  (⋀x. P x ⟹ Q x) ⟹ length (filter P xs) ≤ length (filter Q xs)
```

by (*induction xs*) *auto*

lemma *filter-Un*:
set xs = *A* ∪ *B* ⟹ *set* (*filter P xs*) = { *x* ∈ *A*. *P x* } ∪ { *x* ∈ *B*. *P x* }
by (*induction xs*) (*auto*, *metis UnI1 insert-iff*, *metis UnI2 insert-iff*)

1.1.4 *length*

fun *length-tm* :: *'a list* ⇒ *nat tm* **where**
length-tm [] =*1 return 0*
| *length-tm* (*x* # *xs*) =*1*
do {
l <− *length-tm xs*;
return (*1* + *l*)
}

lemma *length-eq-val-length-tm*:
val (*length-tm xs*) = *length xs*
by (*induction xs*) *auto*

lemma *time-length-tm*:
time (*length-tm xs*) = *length xs* + *1*
by (*induction xs*) (*auto simp*: *time-simps*)

fun *length-it'* :: *nat* ⇒ *'a list* ⇒ *nat* **where**
length-it' acc [] = *acc*
| *length-it' acc* (*x*#*xs*) = *length-it'* (*acc*+*1*) *xs*

definition *length-it* :: *'a list* ⇒ *nat* **where**
length-it xs = *length-it' 0 xs*

lemma *length-conv-length-it'*:
length xs + *acc* = *length-it' acc xs*
by (*induction acc xs rule*: *length-it'.induct*) *auto*

lemma *length-conv-length-it*[*code-unfold*]:
length xs = *length-it xs*
unfolding *length-it-def* **using** *length-conv-length-it' add-0-right* **by** *metis*

1.1.5 *rev*

fun *rev-it'* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
rev-it' acc [] = *acc*
| *rev-it' acc* (*x*#*xs*) = *rev-it'* (*x*#*acc*) *xs*

definition *rev-it* :: *'a list* ⇒ *'a list* **where**
rev-it xs = *rev-it'* [] *xs*

lemma *rev-conv-rev-it'*:
rev xs @ *acc* = *rev-it' acc xs*

by (*induction acc xs rule*: *rev-it'.induct*) *auto*

lemma *rev-conv-rev-it*[*code-unfold*]:
rev xs = *rev-it xs*
unfolding *rev-it-def* **using** *rev-conv-rev-it'* *append-Nil2* **by** *metis*

1.1.6 *take*

fun *take-tm* :: *nat* ⇒ *'a list* ⇒ *'a list tm* **where**
take-tm n [] =*1 return* []
| *take-tm n* (*x* # *xs*) =*1*
(*case n of*
0 ⇒ *return* []
| *Suc m* ⇒ *do* {
ys <− *take-tm m xs*;
return (*x* # *ys*)
}
)

lemma *take-eq-val-take-tm*:
val (*take-tm n xs*) = *take n xs*
by (*induction xs arbitrary*: *n*) (*auto split*: *nat.split*)

lemma *time-take-tm*:
time (*take-tm n xs*) = *min n* (*length xs*) + *1*
by (*induction xs arbitrary*: *n*) (*auto simp*: *time-simps split*: *nat.split*)

1.1.7 *filter*

fun *filter-tm* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list tm* **where**
filter-tm P [] =*1 return* []
| *filter-tm P* (*x* # *xs*) =*1*
(*if P x then*
do {
ys <− *filter-tm P xs*;
return (*x* # *ys*)
}
else
filter-tm P xs
)

lemma *filter-eq-val-filter-tm*:
val (*filter-tm P xs*) = *filter P xs*
by (*induction xs*) *auto*

lemma *time-filter-tm*:
time (*filter-tm P xs*) = *length xs* + *1*
by (*induction xs*) (*auto simp*: *time-simps*)

fun *filter-it'* :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list* **where**

```
  filter-it' acc P [] = rev acc
| filter-it' acc P (x#xs) = (
    if P x then
      filter-it' (x#acc) P xs
    else
      filter-it' acc P xs
  )

definition filter-it :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list where
  filter-it P xs = filter-it' [] P xs

lemma filter-conv-filter-it':
  rev acc @ filter P xs = filter-it' acc P xs
  by (induction acc P xs rule: filter-it'.induct) auto

lemma filter-conv-filter-it[code-unfold]:
  filter P xs = filter-it P xs
   unfolding filter-it-def using filter-conv-filter-it' append-Nil rev.simps(1) by
metis
```

1.1.8 *split-at*

```
fun split-at-tm :: nat ⇒ 'a list ⇒ ('a list × 'a list) tm where
  split-at-tm n [] =1 return ([], [])
| split-at-tm n (x # xs) =1 (
    case n of
      0 ⇒ return ([], x # xs)
    | Suc m ⇒
      do {
        (xs', ys') <- split-at-tm m xs;
        return (x # xs', ys')
      }
  )

fun split-at :: nat ⇒ 'a list ⇒ 'a list × 'a list where
  split-at n [] = ([], [])
| split-at n (x # xs) = (
    case n of
      0 ⇒ ([], x # xs)
    | Suc m ⇒
        let (xs', ys') = split-at m xs in
        (x # xs', ys')
  )

lemma split-at-eq-val-split-at-tm:
  val (split-at-tm n xs) = split-at n xs
  by (induction xs arbitrary: n) (auto split: nat.split prod.split)

lemma split-at-take-drop-conv:
```

```
  split-at n xs = (take n xs, drop n xs)
  by (induction xs arbitrary: n) (auto simp: split: nat.split)

lemma time-split-at-tm:
  time (split-at-tm n xs) = min n (length xs) + 1
  by (induction xs arbitrary: n) (auto simp: time-simps split: nat.split prod.split)

fun split-at-it' :: 'a list ⇒ nat ⇒ 'a list ⇒ ('a list * 'a list) where
  split-at-it' acc n [] = (rev acc, [])
| split-at-it' acc n (x#xs) = (
    case n of
      0 ⇒ (rev acc, x#xs)
    | Suc m ⇒ split-at-it' (x#acc) m xs
  )

definition split-at-it :: nat ⇒ 'a list ⇒ ('a list * 'a list) where
  split-at-it n xs = split-at-it' [] n xs

lemma split-at-conv-split-at-it':
  assumes (ts, ds) = split-at n xs (ts', ds') = split-at-it' acc n xs
  shows rev acc @ ts = ts'
    and ds = ds'
  using assms
  by (induction acc n xs arbitrary: ts rule: split-at-it'.induct)
     (auto simp: split: prod.splits nat.splits)

lemma split-at-conv-split-at-it-prod:
  assumes (ts, ds) = split-at n xs (ts', ds') = split-at-it n xs
  shows (ts, ds) = (ts', ds')
  using assms unfolding split-at-it-def
  using split-at-conv-split-at-it' rev.simps(1) append-Nil by fast+

lemma split-at-conv-split-at-it[code-unfold]:
  split-at n xs = split-at-it n xs
  using split-at-conv-split-at-it-prod surj-pair by metis

declare split-at-tm.simps [simp del]
declare split-at.simps [simp del]
```

1.2 Mergesort

1.2.1 Functional Correctness Proof

```
definition sorted-fst :: point list ⇒ bool where
  sorted-fst ps = sorted-wrt (λp0 p1. fst p0 ≤ fst p1) ps

definition sorted-snd :: point list ⇒ bool where
  sorted-snd ps = sorted-wrt (λp0 p1. snd p0 ≤ snd p1) ps

fun merge-tm :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list ⇒ 'b list tm where
```

```
  merge-tm f (x # xs) (y # ys) =1 (
    if f x ≤ f y then
      do {
        tl <- merge-tm f xs (y # ys);
        return (x # tl)
      }
    else
      do {
        tl <- merge-tm f (x # xs) ys;
        return (y # tl)
      }
  )
| merge-tm f [] ys =1 return ys
| merge-tm f xs [] =1 return xs

fun merge :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list ⇒ 'b list where
  merge f (x # xs) (y # ys) = (
    if f x ≤ f y then
      x # merge f xs (y # ys)
    else
      y # merge f (x # xs) ys
  )
| merge f [] ys = ys
| merge f xs [] = xs

lemma merge-eq-val-merge-tm:
  val (merge-tm f xs ys) = merge f xs ys
  by (induction f xs ys rule: merge.induct) auto

lemma length-merge:
  length (merge f xs ys) = length xs + length ys
  by (induction f xs ys rule: merge.induct) auto

lemma set-merge:
  set (merge f xs ys) = set xs ∪ set ys
  by (induction f xs ys rule: merge.induct) auto

lemma distinct-merge:
  assumes set xs ∩ set ys = {} distinct xs distinct ys
  shows distinct (merge f xs ys)
  using assms by (induction f xs ys rule: merge.induct) (auto simp: set-merge)

lemma sorted-merge:
  assumes P = (λx y. f x ≤ f y)
  shows sorted-wrt P (merge f xs ys) ⟷ sorted-wrt P xs ∧ sorted-wrt P ys
  using assms by (induction f xs ys rule: merge.induct) (auto simp: set-merge)

declare split-at-take-drop-conv [simp]
```

```
function (sequential) mergesort-tm :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list tm
where
  mergesort-tm f [] =1 return []
| mergesort-tm f [x] =1 return [x]
| mergesort-tm f xs =1 (
    do {
      n <− length-tm xs;
      (xs_l, xs_r) <− split-at-tm (n div 2) xs;
      l <− mergesort-tm f xs_l;
      r <− mergesort-tm f xs_r;
      merge-tm f l r
    }
  )
  by pat-completeness auto
termination mergesort-tm
  by (relation Wellfounded.measure (λ(-, xs). length xs))
     (auto simp add: length-eq-val-length-tm split-at-eq-val-split-at-tm)

fun mergesort :: ('b ⇒ 'a::linorder) ⇒ 'b list ⇒ 'b list where
  mergesort f [] = []
| mergesort f [x] = [x]
| mergesort f xs = (
    let n = length xs div 2 in
    let (l, r) = split-at n xs in
    merge f (mergesort f l) (mergesort f r)
  )

declare split-at-take-drop-conv [simp del]

lemma mergesort-eq-val-mergesort-tm:
  val (mergesort-tm f xs) = mergesort f xs
  by (induction f xs rule: mergesort.induct)
    (auto simp add: length-eq-val-length-tm split-at-eq-val-split-at-tm merge-eq-val-merge-tm
split: prod.split)

lemma sorted-wrt-mergesort:
  sorted-wrt (λx y. f x ≤ f y) (mergesort f xs)
   by (induction f xs rule: mergesort.induct) (auto simp: split-at-take-drop-conv
sorted-merge)

lemma set-mergesort:
  set (mergesort f xs) = set xs
  by (induction f xs rule: mergesort.induct)
    (simp-all add: set-merge split-at-take-drop-conv, metis list.simps(15) set-take-drop)

lemma length-mergesort:
  length (mergesort f xs) = length xs
 by (induction f xs rule: mergesort.induct) (auto simp: length-merge split-at-take-drop-conv)
```

lemma *distinct-mergesort*:
distinct xs $\Longrightarrow$ *distinct* (*mergesort f xs*)
proof (*induction f xs rule*: *mergesort.induct*)
case (*3 f x y xs*)
let *?xs'* = *x # y # xs*
obtain *l r* **where** *lr-def*: (*l*, *r*) = *split-at* (*length ?xs' div 2*) *?xs'*
by (*metis surj-pair*)
have *distinct l distinct r*
using *3.prems split-at-take-drop-conv distinct-take distinct-drop lr-def* **by** (*metis prod.sel*)+
hence *distinct* (*mergesort f l*) *distinct* (*mergesort f r*)
using *3.IH lr-def* **by** *auto*
moreover have *set l* $\cap$ *set r* = {}
using *3.prems split-at-take-drop-conv lr-def* **by** (*metis append-take-drop-id distinct-append prod.sel*)
ultimately show *?case*
using *lr-def* **by** (*auto simp*: *distinct-merge set-mergesort split*: *prod.splits*)
qed *auto*

lemmas *mergesort* = *sorted-wrt-mergesort set-mergesort length-mergesort distinct-mergesort*

lemma *sorted-fst-take-less-hd-drop*:
assumes *sorted-fst ps n* < *length ps*
shows $\forall$ *p* $\in$ *set* (*take n ps*). *fst p* $\leq$ *fst* (*hd* (*drop n ps*))
using *assms sorted-wrt-take-less-hd-drop*[*of* $\lambda p_0\ p_1$. *fst* $p_0 \leq$ *fst* p_1] *sorted-fst-def*
by *fastforce*

lemma *sorted-fst-hd-drop-less-drop*:
assumes *sorted-fst ps*
shows $\forall$ *p* $\in$ *set* (*drop n ps*). *fst* (*hd* (*drop n ps*)) $\leq$ *fst p*
using *assms sorted-wrt-hd-drop-less-drop*[*of* $\lambda p_0\ p_1$. *fst* $p_0 \leq$ *fst* p_1] *sorted-fst-def*
by *fastforce*

1.2.2 Time Complexity Proof

lemma *time-merge-tm*:
time (*merge-tm f xs ys*) $\leq$ *length xs* + *length ys* + *1*
by (*induction f xs ys rule*: *merge-tm.induct*) (*auto simp*: *time-simps*)

function *mergesort-recurrence* :: *nat* $\Rightarrow$ *real* **where**
mergesort-recurrence 0 = *1*
| *mergesort-recurrence 1* = *1*
| *2* $\leq$ *n* $\Longrightarrow$ *mergesort-recurrence n* = *4* + *3* $*$ *n* + *mergesort-recurrence* (*nat* $\lfloor$*real n / 2*$\rfloor$) +
mergesort-recurrence (*nat* $\lceil$*real n / 2*$\rceil$)
by *force simp-all*
termination by *akra-bazzi-termination simp-all*

lemma *mergesort-recurrence-nonneg*[*simp*]:

```
  0 ≤ mergesort-recurrence n
  by (induction n rule: mergesort-recurrence.induct) (auto simp del: One-nat-def)

lemma time-mergesort-conv-mergesort-recurrence:
  time (mergesort-tm f xs) ≤ mergesort-recurrence (length xs)
proof (induction f xs rule: mergesort-tm.induct)
  case (1 f)
  thus ?case by (auto simp: time-simps)
next
  case (2 f x)
  thus ?case using mergesort-recurrence.simps(2) by (auto simp: time-simps)
next
  case (3 f x y xs')

  define xs where xs = x # y # xs'
  define n where n = length xs
  obtain l r where lr-def: (l, r) = split-at (n div 2) xs
    using prod.collapse by blast
  define l' where l' = mergesort f l
  define r' where r' = mergesort f r
  note defs = xs-def n-def lr-def l'-def r'-def

  have IHL: time (mergesort-tm f l) ≤ mergesort-recurrence (length l)
   using defs 3.IH(1) by (auto simp: length-eq-val-length-tm split-at-eq-val-split-at-tm)
  have IHR: time (mergesort-tm f r) ≤ mergesort-recurrence (length r)
   using defs 3.IH(2) by (auto simp: length-eq-val-length-tm split-at-eq-val-split-at-tm)

  have *: length l = n div 2 length r = n - n div 2
    using defs by (auto simp: split-at-take-drop-conv)
  hence (nat ⌊real n / 2⌋) = length l (nat ⌈real n / 2⌉) = length r
    by linarith+
  hence IH: time (mergesort-tm f l) ≤ mergesort-recurrence (nat ⌊real n / 2⌋)
          time (mergesort-tm f r) ≤ mergesort-recurrence (nat ⌈real n / 2⌉)
    using IHL IHR by simp-all

  have n = length l + length r
    using * by linarith
  hence time (merge-tm f l' r') ≤ n + 1
    using time-merge-tm defs by (metis length-mergesort)

  have time (mergesort-tm f xs) = 1 + time (length-tm xs) + time (split-at-tm (n
div 2) xs) +
        time (mergesort-tm f l) + time (mergesort-tm f r) + time (merge-tm f l'
r')
     using defs by (auto simp add: time-simps length-eq-val-length-tm  merge-
sort-eq-val-mergesort-tm
                                   split-at-eq-val-split-at-tm
                         split: prod.split)
  also have ... ≤ 4 + 3 * n + time (mergesort-tm f l) + time (mergesort-tm f r)
```

using *time-length-tm*[*of xs*] *time-split-at-tm*[*of n div 2 xs*] *n-def* ‹*time* (*merge-tm f l′ r′*) ≤ *n* + *1*› **by** *simp*
also have ... ≤ *4* + *3* ∗ *n* + *mergesort-recurrence* (*nat* ⌊*real n* / *2*⌋) + *mergesort-recurrence* (*nat* ⌈*real n* / *2*⌉)
using *IH* **by** *simp*
also have ... = *mergesort-recurrence n*
using *defs* **by** *simp*
finally show *?case*
using *defs* **by** *simp*
qed

theorem *mergesort-recurrence*:
mergesort-recurrence ∈ Θ(λ*n*. *n* ∗ *ln n*)
by (*master-theorem*) *auto*

theorem *time-mergesort-tm-bigo*:
(λ*xs*. *time* (*mergesort-tm f xs*)) ∈ *O*[*length going-to at-top*]((λ*n*. *n* ∗ *ln n*) *o length*)
proof −
have *0*: ⋀*xs*. *time* (*mergesort-tm f xs*) ≤ (*mergesort-recurrence o length*) *xs*
unfolding *comp-def* **using** *time-mergesort-conv-mergesort-recurrence* **by** *blast*
show *?thesis*
using *bigo-measure-trans*[*OF 0*] **by** (*simp add*: *bigthetaD1 mergesort-recurrence*)
qed

1.2.3 Code Export

lemma *merge-xs-Nil*[*simp*]:
merge f xs [] = *xs*
by (*cases xs*) *auto*

fun *merge-it′* :: (′*b* ⇒ ′*a*::*linorder*) ⇒ ′*b list* ⇒ ′*b list* ⇒ ′*b list* ⇒ ′*b list* **where**
merge-it′ f acc [] [] = *rev acc*
| *merge-it′ f acc* (*x*#*xs*) [] = *merge-it′ f* (*x*#*acc*) *xs* []
| *merge-it′ f acc* [] (*y*#*ys*) = *merge-it′ f* (*y*#*acc*) *ys* []
| *merge-it′ f acc* (*x*#*xs*) (*y*#*ys*) = (
if f x ≤ *f y then*
merge-it′ f (*x*#*acc*) *xs* (*y*#*ys*)
else
merge-it′ f (*y*#*acc*) (*x*#*xs*) *ys*
)

definition *merge-it* :: (′*b* ⇒ ′*a*::*linorder*) ⇒ ′*b list* ⇒ ′*b list* ⇒ ′*b list* **where**
merge-it f xs ys = *merge-it′ f* [] *xs ys*

lemma *merge-conv-merge-it′*:
rev acc @ *merge f xs ys* = *merge-it′ f acc xs ys*
by (*induction f acc xs ys rule*: *merge-it′.induct*) *auto*

lemma *merge-conv-merge-it*[*code-unfold*]:

merge f xs ys = merge-it f xs ys
unfolding *merge-it-def* **using** *merge-conv-merge-it' rev.simps(1) append-Nil* **by** *metis*

1.3 Minimal Distance

definition *sparse* :: *real* ⇒ *point set* ⇒ *bool* **where**
sparse δ ps ⟷ (∀ *p*$_0$ ∈ *ps*. ∀ *p*$_1$ ∈ *ps*. *p*$_0$ ≠ *p*$_1$ ⟶ *δ* ≤ *dist p*$_0$ *p*$_1$)

lemma *sparse-identity*:
assumes *sparse δ* (*set ps*) ∀ *p* ∈ *set ps*. *δ* ≤ *dist p*$_0$ *p*
shows *sparse δ* (*set* (*p*$_0$ # *ps*))
using *assms* **by** (*simp add*: *dist-commute sparse-def*)

lemma *sparse-update*:
assumes *sparse δ* (*set ps*)
assumes *dist p*$_0$ *p*$_1$ ≤ *δ* ∀ *p* ∈ *set ps*. *dist p*$_0$ *p*$_1$ ≤ *dist p*$_0$ *p*
shows *sparse* (*dist p*$_0$ *p*$_1$) (*set* (*p*$_0$ # *ps*))
using *assms* **by** (*auto simp*: *dist-commute sparse-def*, *force*+)

lemma *sparse-mono*:
sparse Δ P ⟹ *δ* ≤ *Δ* ⟹ *sparse δ P*
unfolding *sparse-def* **by** *fastforce*

1.4 Distance

lemma *dist-transform*:
fixes *p* :: *point* **and** *δ* :: *real* **and** *l* :: *int*
shows *dist p* (*l*, *snd p*) < *δ* ⟷ *l* − *δ* < *fst p* ∧ *fst p* < *l* + *δ*
proof −
have *dist p* (*l*, *snd p*) = *sqrt* ((*real-of-int* (*fst p*) − *l*)^2)
by (*auto simp add*: *dist-prod-def dist-real-def prod.case-eq-if*)
thus *?thesis*
by *auto*
qed

fun *dist-code* :: *point* ⇒ *point* ⇒ *int* **where**
dist-code p$_0$ *p*$_1$ = (*fst p*$_0$ − *fst p*$_1$)^2 + (*snd p*$_0$ − *snd p*$_1$)^2

lemma *dist-eq-sqrt-dist-code*:
fixes *p*$_0$ :: *point*
shows *dist p*$_0$ *p*$_1$ = *sqrt* (*dist-code p*$_0$ *p*$_1$)
by (*auto simp*: *dist-prod-def dist-real-def split*: *prod.splits*)

lemma *dist-eq-dist-code-lt*:
fixes *p*$_0$ :: *point*
shows *dist p*$_0$ *p*$_1$ < *dist p*$_2$ *p*$_3$ ⟷ *dist-code p*$_0$ *p*$_1$ < *dist-code p*$_2$ *p*$_3$
using *dist-eq-sqrt-dist-code real-sqrt-less-iff* **by** *presburger*

lemma *dist-eq-dist-code-le*:

fixes p_0 :: *point*
shows *dist* p_0 $p_1 \leq$ *dist* p_2 $p_3 \longleftrightarrow$ *dist-code* p_0 $p_1 \leq$ *dist-code* p_2 p_3
using *dist-eq-sqrt-dist-code real-sqrt-le-iff* **by** *presburger*

lemma *dist-eq-dist-code-abs-lt*:
fixes p_0 :: *point*
shows $|c| <$ *dist* p_0 $p_1 \longleftrightarrow c^2 <$ *dist-code* p_0 p_1
using *dist-eq-sqrt-dist-code*
by (*metis of-int-less-of-int-power-cancel-iff real-sqrt-abs real-sqrt-less-iff*)

lemma *dist-eq-dist-code-abs-le*:
fixes p_0 :: *point*
shows *dist* p_0 $p_1 \leq |c| \longleftrightarrow$ *dist-code* p_0 $p_1 \leq c^2$
using *dist-eq-sqrt-dist-code*
by (*metis of-int-power-le-of-int-cancel-iff real-sqrt-abs real-sqrt-le-iff*)

lemma *dist-fst-abs*:
fixes p :: *point* **and** l :: *int*
shows *dist* p (l, *snd* p) = |*fst* p − l|
proof −
have *dist* p (l, *snd* p) = *sqrt* ((*real-of-int* (*fst* p) − l)^2)
by (*simp add*: *dist-prod-def dist-real-def prod.case-eq-if*)
thus *?thesis*
by *simp*
qed

declare *dist-code.simps* [*simp del*]

1.5 Brute Force Closest Pair Algorithm

1.5.1 Functional Correctness Proof

```
fun find-closest-bf-tm :: point ⇒ point list ⇒ point tm where
  find-closest-bf-tm - [] =1 return undefined
| find-closest-bf-tm - [p] =1 return p
| find-closest-bf-tm p (p₀ # ps) =1 (
    do {
      p₁ <− find-closest-bf-tm p ps;
      if dist p p₀ < dist p p₁ then
        return p₀
      else
        return p₁
    }
  )

fun find-closest-bf :: point ⇒ point list ⇒ point where
  find-closest-bf - [] = undefined
| find-closest-bf - [p] = p
| find-closest-bf p (p₀ # ps) = (
    let p₁ = find-closest-bf p ps in
```

```
    if dist p p₀ < dist p p₁ then
      p₀
    else
      p₁
  )

lemma find-closest-bf-eq-val-find-closest-bf-tm:
  val (find-closest-bf-tm p ps) = find-closest-bf p ps
  by (induction p ps rule: find-closest-bf.induct) (auto simp: Let-def)

lemma find-closest-bf-set:
  0 < length ps ⟹ find-closest-bf p ps ∈ set ps
  by (induction p ps rule: find-closest-bf.induct)
     (auto simp: Let-def split: prod.splits if-splits)

lemma find-closest-bf-dist:
  ∀ q ∈ set ps. dist p (find-closest-bf p ps) ≤ dist p q
  by (induction p ps rule: find-closest-bf.induct)
     (auto split: prod.splits)

fun closest-pair-bf-tm :: point list ⇒ (point × point) tm where
  closest-pair-bf-tm [] =1 return undefined
| closest-pair-bf-tm [-] =1 return undefined
| closest-pair-bf-tm [p₀, p₁] =1 return (p₀, p₁)
| closest-pair-bf-tm (p₀ # ps) =1 (
    do {
      (c₀::point, c₁::point) <− closest-pair-bf-tm ps;
      p₁ <− find-closest-bf-tm p₀ ps;
      if dist c₀ c₁ ≤ dist p₀ p₁ then
        return (c₀, c₁)
      else
        return (p₀, p₁)
    }
  )

fun closest-pair-bf :: point list ⇒ (point ∗ point) where
  closest-pair-bf [] = undefined
| closest-pair-bf [-] = undefined
| closest-pair-bf [p₀, p₁] = (p₀, p₁)
| closest-pair-bf (p₀ # ps) = (
    let (c₀, c₁) = closest-pair-bf ps in
    let p₁ = find-closest-bf p₀ ps in
    if dist c₀ c₁ ≤ dist p₀ p₁ then
      (c₀, c₁)
    else
      (p₀, p₁)
  )

lemma closest-pair-bf-eq-val-closest-pair-bf-tm:
```

```
  val (closest-pair-bf-tm ps) = closest-pair-bf ps
  by (induction ps rule: closest-pair-bf.induct)
    (auto simp: Let-def find-closest-bf-eq-val-find-closest-bf-tm split: prod.split)

lemma closest-pair-bf-c0:
  1 < length ps ⟹ (c0, c1) = closest-pair-bf ps ⟹ c0 ∈ set ps
  by (induction ps arbitrary: c0 c1 rule: closest-pair-bf.induct)
    (auto simp: Let-def find-closest-bf-set split: if-splits prod.splits)

lemma closest-pair-bf-c1:
  1 < length ps ⟹ (c0, c1) = closest-pair-bf ps ⟹ c1 ∈ set ps
proof (induction ps arbitrary: c0 c1 rule: closest-pair-bf.induct)
  case (4 p0 p2 p3 ps)
  let ?ps = p2 # p3 # ps
  obtain c0 c1 where c0-def: (c0, c1) = closest-pair-bf ?ps
    using prod.collapse by blast
  define p1 where p1-def: p1 = find-closest-bf p0 ?ps
  note defs = c0-def p1-def
  have c1 ∈ set ?ps
    using 4.IH defs by simp
  moreover have p1 ∈ set ?ps
    using find-closest-bf-set defs by blast
  ultimately show ?case
    using 4.prems(2) defs by (auto simp: Let-def split: prod.splits if-splits)
qed auto

lemma closest-pair-bf-c0-ne-c1:
  1 < length ps ⟹ distinct ps ⟹ (c0, c1) = closest-pair-bf ps ⟹ c0 ≠ c1
proof (induction ps arbitrary: c0 c1 rule: closest-pair-bf.induct)
  case (4 p0 p2 p3 ps)
  let ?ps = p2 # p3 # ps
  obtain c0 c1 where c0-def: (c0, c1) = closest-pair-bf ?ps
    using prod.collapse by blast
  define p1 where p1-def: p1 = find-closest-bf p0 ?ps
  note defs = c0-def p1-def
  have c0 ≠ c1
    using 4.IH 4.prems(2) defs by simp
  moreover have p0 ≠ p1
    using find-closest-bf-set 4.prems(2) defs
    by (metis distinct.simps(2) length-pos-if-in-set list.set-intros(1))
  ultimately show ?case
    using 4.prems(3) defs by (auto simp: Let-def split: prod.splits if-splits)
qed auto

lemmas closest-pair-bf-c0-c1 = closest-pair-bf-c0 closest-pair-bf-c1 closest-pair-bf-c0-ne-c1

lemma closest-pair-bf-dist:
  assumes 1 < length ps (c0, c1) = closest-pair-bf ps
  shows sparse (dist c0 c1) (set ps)
```

using *assms*
proof (*induction ps arbitrary*: c_0 c_1 *rule*: *closest-pair-bf.induct*)
case (*4* p_0 p_2 p_3 *ps*)
let *?ps* = p_2 # p_3 # *ps*
obtain c_0 c_1 **where** c_0*-def*: (c_0, c_1) = *closest-pair-bf ?ps*
using *prod.collapse* **by** *blast*
define p_1 **where** p_1*-def*: p_1 = *find-closest-bf* p_0 *?ps*
note *defs* = c_0*-def* p_1*-def*
hence *IH*: *sparse* (*dist* c_0 c_1) (*set ?ps*)
using *4* c_0*-def* **by** *simp*
have ∗: ∀ *p* ∈ *set ?ps*. (*dist* p_0 p_1) ≤ *dist* p_0 *p*
using *find-closest-bf-dist defs* **by** *blast*
show *?case*
proof (*cases dist* c_0 c_1 ≤ *dist* p_0 p_1)
case *True*
hence ∀ *p* ∈ *set ?ps*. *dist* c_0 c_1 ≤ *dist* p_0 *p*
using ∗ **by** *auto*
hence *sparse* (*dist* c_0 c_1) (*set* (p_0 # *?ps*))
using *sparse-identity IH* **by** *blast*
thus *?thesis*
using *True 4.prems defs* **by** (*auto split*: *prod.splits*)
next
case *False*
hence *sparse* (*dist* p_0 p_1) (*set* (p_0 # *?ps*))
using *sparse-update*[*of dist* c_0 c_1 *?ps* p_0 p_1] *IH* ∗ *defs* **by** *argo*
thus *?thesis*
using *False 4.prems defs* **by** (*auto split*: *prod.splits*)
qed
qed (*auto simp*: *dist-commute sparse-def*)

1.5.2 Time Complexity Proof

lemma *time-find-closest-bf-tm*:
time (*find-closest-bf-tm p ps*) ≤ *length ps* + *1*
by (*induction p ps rule*: *find-closest-bf-tm.induct*) (*auto simp*: *time-simps*)

lemma *time-closest-pair-bf-tm*:
time (*closest-pair-bf-tm ps*) ≤ *length ps* ∗ *length ps* + *1*
proof (*induction ps rule*: *closest-pair-bf-tm.induct*)
case (*4* p_0 p_2 p_3 *ps*)
let *?ps* = p_2 # p_3 # *ps*
have *time* (*closest-pair-bf-tm* (p_0 # *?ps*)) = *1* + *time* (*find-closest-bf-tm* p_0 *?ps*)
+ *time* (*closest-pair-bf-tm ?ps*)
by (*auto simp*: *time-simps split*: *prod.split*)
also have ... ≤ *2* + *length ?ps* + *time* (*closest-pair-bf-tm ?ps*)
using *time-find-closest-bf-tm*[*of* p_0 *?ps*] **by** *simp*
also have ... ≤ *2* + *length ?ps* + *length ?ps* ∗ *length ?ps* + *1*
using *4.IH* **by** *simp*
also have ... ≤ *length* (p_0 # *?ps*) ∗ *length* (p_0 # *?ps*) + *1*

by *auto*
finally show *?case*
by *blast*
qed (*auto simp*: *time-simps*)

1.5.3 Code Export

fun *find-closest-bf-code* :: *point* ⇒ *point list* ⇒ (*int* ∗ *point*) **where**
find-closest-bf-code p [] = *undefined*
| *find-closest-bf-code p* [p_0] = (*dist-code p* p_0, p_0)
| *find-closest-bf-code p* (p_0 # *ps*) = (
let (δ_1, p_1) = *find-closest-bf-code p ps in*
let δ_0 = *dist-code p* p_0 *in*
if $\delta_0 < \delta_1$ *then*
(δ_0, p_0)
else
(δ_1, p_1)
)

lemma *find-closest-bf-code-dist-eq*:
0 < length ps ⟹ (δ, *c*) = *find-closest-bf-code p ps* ⟹ δ = *dist-code p c*
by (*induction p ps rule*: *find-closest-bf-code.induct*)
(*auto simp*: *Let-def split*: *prod.splits if-splits*)

lemma *find-closest-bf-code-eq*:
0 < length ps ⟹ *c* = *find-closest-bf p ps* ⟹ (δ', *c'*) = *find-closest-bf-code p ps*
⟹ *c* = *c'*
proof (*induction p ps arbitrary*: *c* δ' *c' rule*: *find-closest-bf.induct*)
case (*3 p* p_0 p_2 *ps*)
define δ_0 δ_0' **where** δ_0-*def*: δ_0 = *dist p* p_0 δ_0' = *dist-code p* p_0
obtain δ_1 p_1 δ_1' p_1' **where** δ_1-*def*: δ_1 = *dist p* p_1 p_1 = *find-closest-bf p* (p_2 # *ps*)
(δ_1', p_1') = *find-closest-bf-code p* (p_2 # *ps*)
using *prod.collapse* **by** *blast+*
note *defs* = δ_0-*def* δ_1-*def*
have ∗: p_1 = p_1'
using *3.IH defs* **by** *simp*
hence $\delta_0 < \delta_1 \longleftrightarrow \delta_0' < \delta_1'$
using *find-closest-bf-code-dist-eq*[*of* p_2 # *ps* δ_1' p_1' *p*]
dist-eq-dist-code-lt defs
by *simp*
thus *?case*
using *3.prems*(*2*,*3*) ∗ *defs* **by** (*auto split*: *prod.splits*)
qed *auto*

declare *find-closest-bf-code.simps* [*simp del*]

fun *closest-pair-bf-code* :: *point list* ⇒ (*int* ∗ *point* ∗ *point*) **where**
closest-pair-bf-code [] = *undefined*

```
| closest-pair-bf-code [p0] = undefined
| closest-pair-bf-code [p0, p1] = (dist-code p0 p1, p0, p1)
| closest-pair-bf-code (p0 # ps) = (
    let (δc, c0, c1) = closest-pair-bf-code ps in
    let (δp, p1) = find-closest-bf-code p0 ps in
    if δc ≤ δp then
      (δc, c0, c1)
    else
      (δp, p0, p1)
  )

lemma closest-pair-bf-code-dist-eq:
  1 < length ps ⟹ (δ, c0, c1) = closest-pair-bf-code ps ⟹ δ = dist-code c0 c1
proof (induction ps arbitrary: δ c0 c1 rule: closest-pair-bf-code.induct)
  case (4 p0 p2 p3 ps)
  let ?ps = p2 # p3 # ps
  obtain δc c0 c1 where δc-def: (δc, c0, c1) = closest-pair-bf-code ?ps
    by (metis prod-cases3)
  obtain δp p1 where δp-def: (δp, p1) = find-closest-bf-code p0 ?ps
    using prod.collapse by blast
  note defs = δc-def δp-def
  have δc = dist-code c0 c1
    using 4.IH defs by simp
  moreover have δp = dist-code p0 p1
    using find-closest-bf-code-dist-eq defs by blast
  ultimately show ?case
    using 4.prems(2) defs by (auto split: prod.splits if-splits)
qed auto

lemma closest-pair-bf-code-eq:
  assumes 1 < length ps
  assumes (c0, c1) = closest-pair-bf ps (δ', c0', c1') = closest-pair-bf-code ps
  shows c0 = c0' ∧ c1 = c1'
  using assms
proof (induction ps arbitrary: c0 c1 δ' c0' c1' rule: closest-pair-bf-code.induct)
  case (4 p0 p2 p3 ps)
  let ?ps = p2 # p3 # ps
  obtain c0 c1 δc' c0' c1' where δc-def: (c0, c1) = closest-pair-bf ?ps
    (δc', c0', c1') = closest-pair-bf-code ?ps
    by (metis prod-cases3)
  obtain p1 δp' p1' where δp-def: p1 = find-closest-bf p0 ?ps
    (δp', p1') = find-closest-bf-code p0 ?ps
    using prod.collapse by blast
  note defs = δc-def δp-def
  have A: c0 = c0' ∧ c1 = c1'
    using 4.IH defs by simp
  moreover have B: p1 = p1'
    using find-closest-bf-code-eq defs by blast
  moreover have δc' = dist-code c0' c1'
```

using *defs closest-pair-bf-code-dist-eq*[*of ?ps*] **by** *simp*
moreover have $\delta_p{}' = dist\text{-}code\ p_0\ p_1{}'$
using *defs find-closest-bf-code-dist-eq* **by** *blast*
ultimately have $dist\ c_0\ c_1 \leq dist\ p_0\ p_1 \longleftrightarrow \delta_c{}' \leq \delta_p{}'$
by (*simp add*: *dist-eq-dist-code-le*)
thus *?case*
using *4.prems*(*2*,*3*) *defs A B* **by** (*auto simp*: *Let-def split*: *prod.splits*)
qed *auto*

1.6 Geometry

1.6.1 Band Filter

lemma *set-band-filter-aux*:
fixes δ :: *real* **and** *ps* :: *point list*
assumes $p_0 \in ps_L\ p_1 \in ps_R\ p_0 \neq p_1\ dist\ p_0\ p_1 < \delta\ set\ ps = ps_L \cup ps_R$
assumes $\forall p \in ps_L.\ fst\ p \leq l\ \forall p \in ps_R.\ l \leq fst\ p$
assumes $ps' = filter\ (\lambda p.\ l - \delta < fst\ p \wedge fst\ p < l + \delta)\ ps$
shows $p_0 \in set\ ps' \wedge p_1 \in set\ ps'$
proof (*rule ccontr*)
assume $\neg\ (p_0 \in set\ ps' \wedge p_1 \in set\ ps')$
then consider $(A)\ p_0 \notin set\ ps' \wedge p_1 \notin set\ ps'$
$\mid (B)\ p_0 \in set\ ps' \wedge p_1 \notin set\ ps'$
$\mid (C)\ p_0 \notin set\ ps' \wedge p_1 \in set\ ps'$
by *blast*
thus *False*
proof *cases*
case *A*
hence $fst\ p_0 \leq l - \delta \vee l + \delta \leq fst\ p_0\ fst\ p_1 \leq l - \delta \vee l + \delta \leq fst\ p_1$
using *assms*(*1*,*2*,*5*,*8*) **by** *auto*
hence $fst\ p_0 \leq l - \delta\ l + \delta \leq fst\ p_1$
using *assms*(*1*,*2*,*6*,*7*) **by** *force+*
hence $\delta \leq dist\ (fst\ p_0)\ (fst\ p_1)$
using *dist-real-def* **by** *simp*
hence $\delta \leq dist\ p_0\ p_1$
using *dist-fst-le*[*of* $p_0\ p_1$] **by** (*auto split*: *prod.splits*)
then show *?thesis*
using *assms*(*4*) **by** *fastforce*
next
case *B*
hence $fst\ p_1 \leq l - \delta \vee l + \delta \leq fst\ p_1$
using *assms*(*2*,*5*,*8*) **by** *auto*
hence $l + \delta \leq fst\ p_1$
using *assms*(*2*,*7*) **by** *auto*
moreover have $fst\ p_0 \leq l$
using *assms*(*1*,*6*) **by** *simp*
ultimately have $\delta \leq dist\ (fst\ p_0)\ (fst\ p_1)$
using *dist-real-def* **by** *simp*
hence $\delta \leq dist\ p_0\ p_1$
using *dist-fst-le*[*of* $p_0\ p_1$] *less-le-trans* **by** (*auto split*: *prod.splits*)

```
    thus ?thesis
      using assms(4) by simp
  next
    case C
    hence fst p₀ ≤ l − δ ∨ l + δ ≤ fst p₀
      using assms(1,2,5,8) by auto
    hence fst p₀ ≤ l − δ
      using assms(1,6) by auto
    moreover have l ≤ fst p₁
      using assms(2,7) by simp
    ultimately have δ ≤ dist (fst p₀) (fst p₁)
      using dist-real-def by simp
    hence δ ≤ dist p₀ p₁
      using dist-fst-le[of p₀ p₁] less-le-trans by (auto split: prod.splits)
    thus ?thesis
      using assms(4) by simp
  qed
qed

lemma set-band-filter:
  fixes δ :: real and ps :: point list
  assumes p₀ ∈ set ps p₁ ∈ set ps p₀ ≠ p₁ dist p₀ p₁ < δ set ps = ps_L ∪ ps_R
  assumes sparse δ ps_L sparse δ ps_R
  assumes ∀ p ∈ ps_L. fst p ≤ l ∀ p ∈ ps_R. l ≤ fst p
  assumes ps' = filter (λp. l − δ < fst p ∧ fst p < l + δ) ps
  shows p₀ ∈ set ps' ∧ p₁ ∈ set ps'
proof −
  have p₀ ∉ ps_L ∨ p₁ ∉ ps_L p₀ ∉ ps_R ∨ p₁ ∉ ps_R
    using assms(3,4,6,7) sparse-def by force+
  then consider (A) p₀ ∈ ps_L ∧ p₁ ∈ ps_R | (B) p₀ ∈ ps_R ∧ p₁ ∈ ps_L
    using assms(1,2,5) by auto
  thus ?thesis
  proof cases
    case A
    thus ?thesis
      using set-band-filter-aux assms(3,4,5,8,9,10) by (auto split: prod.splits)
  next
    case B
    moreover have dist p₁ p₀ < δ
      using assms(4) dist-commute by metis
    ultimately show ?thesis
     using set-band-filter-aux assms(3)[symmetric] assms(5,8,9,10) by (auto split:
prod.splits)
  qed
qed
```

1.6.2 2D-Boxes and Points

lemma *cbox-2D*:

fixes x_0 :: *real* **and** y_0 :: *real*
shows *cbox* (x_0, y_0) (x_1, y_1) = { (x, y). $x_0 \leq x \wedge x \leq x_1 \wedge y_0 \leq y \wedge y \leq y_1$ }
by *fastforce*

lemma *mem-cbox-2D*:
fixes x :: *real* **and** y :: *real*
shows $x_0 \leq x \wedge x \leq x_1 \wedge y_0 \leq y \wedge y \leq y_1 \longleftrightarrow (x, y) \in$ *cbox* (x_0, y_0) (x_1, y_1)
by *fastforce*

lemma *cbox-top-un*:
fixes x_0 :: *real* **and** y_0 :: *real*
assumes $y_0 \leq y_1$ $y_1 \leq y_2$
shows *cbox* (x_0, y_0) (x_1, y_1) $\cup$ *cbox* (x_0, y_1) (x_1, y_2) = *cbox* (x_0, y_0) (x_1, y_2)
using *assms* **by** *auto*

lemma *cbox-right-un*:
fixes x_0 :: *real* **and** y_0 :: *real*
assumes $x_0 \leq x_1$ $x_1 \leq x_2$
shows *cbox* (x_0, y_0) (x_1, y_1) $\cup$ *cbox* (x_1, y_0) (x_2, y_1) = *cbox* (x_0, y_0) (x_2, y_1)
using *assms* **by** *auto*

lemma *cbox-max-dist*:
assumes $p_0 = (x, y)$ $p_1 = (x + \delta, y + \delta)$
assumes $(x_0, y_0) \in$ *cbox* p_0 p_1 $(x_1, y_1) \in$ *cbox* p_0 p_1 $0 \leq \delta$
shows *dist* (x_0, y_0) $(x_1, y_1) \leq$ *sqrt* $2 * \delta$
proof −
have X: *dist* x_0 $x_1 \leq \delta$
using *assms dist-real-def* **by** *auto*
have Y: *dist* y_0 $y_1 \leq \delta$
using *assms dist-real-def* **by** *auto*

have *dist* (x_0, y_0) (x_1, y_1) = *sqrt* $((dist\ x_0\ x_1)^2 + (dist\ y_0\ y_1)^2)$
using *dist-Pair-Pair* **by** *auto*
also have ... $\leq$ *sqrt* $(\delta^2 + (dist\ y_0\ y_1)^2)$
using X *power-mono* **by** *fastforce*
also have ... $\leq$ *sqrt* $(\delta^2 + \delta^2)$
using Y *power-mono* **by** *fastforce*
also have ... = *sqrt* $2 *$ *sqrt* (δ^2)
using *real-sqrt-mult* **by** *simp*
also have ... = *sqrt* $2 * \delta$
using *assms*(5) **by** *simp*
finally show *?thesis* **.**
qed

1.6.3 Pigeonhole Argument

lemma *card-le-1-if-pairwise-eq*:
assumes $\forall\, x \in S.\ \forall\, y \in S.\ x = y$
shows *card* $S \leq 1$

proof (*rule ccontr*)
assume ¬ *card S* ≤ *1*
hence *2* ≤ *card S*
by *simp*
then obtain *T* **where** ∗: *T* ⊆ *S* ∧ *card T* = *2*
using *ex-card* **by** *metis*
then obtain *x y* **where** *x* ∈ *T* ∧ *y* ∈ *T* ∧ *x* ≠ *y*
by (*meson card-2-iff'*)
then show *False*
using ∗ *assms* **by** *blast*
qed

lemma *card-Int-if-either-in*:
assumes ∀ *x* ∈ *S*. ∀ *y* ∈ *S*. *x* = *y* ∨ *x* ∉ *T* ∨ *y* ∉ *T*
shows *card* (*S* ∩ *T*) ≤ *1*
proof (*rule ccontr*)
assume ¬ (*card* (*S* ∩ *T*) ≤ *1*)
then obtain *x y* **where** ∗: *x* ∈ *S* ∩ *T* ∧ *y* ∈ *S* ∩ *T* ∧ *x* ≠ *y*
by (*meson card-le-1-if-pairwise-eq*)
hence *x* ∈ *T y* ∈ *T*
by *simp-all*
moreover have *x* ∉ *T* ∨ *y* ∉ *T*
using *assms* ∗ **by** *auto*
ultimately show *False*
by *blast*
qed

lemma *card-Int-Un-le-Sum-card-Int*:
assumes *finite S*
shows *card* (*A* ∩ ⋃ *S*) ≤ (∑ *B* ∈ *S*. *card* (*A* ∩ *B*))
using *assms*
proof (*induction card S arbitrary*: *S*)
case (*Suc n*)
then obtain *B T* **where** ∗: *S* = { *B* } ∪ *T card T* = *n B* ∉ *T*
by (*metis card-Suc-eq Suc-eq-plus1 insert-is-Un*)
hence *card* (*A* ∩ ⋃ *S*) = *card* (*A* ∩ ⋃ ({ *B* } ∪ *T*))
by *blast*
also have ... ≤ *card* (*A* ∩ *B*) + *card* (*A* ∩ ⋃ *T*)
by (*simp add*: *card-Un-le inf-sup-distrib1*)
also have ... ≤ *card* (*A* ∩ *B*) + (∑ *B* ∈ *T*. *card* (*A* ∩ *B*))
using *Suc* ∗ **by** *simp*
also have ... ≤ (∑ *B* ∈ *S*. *card* (*A* ∩ *B*))
using *Suc.prems* ∗ **by** *simp*
finally show *?case* **.**
qed *simp*

lemma *pigeonhole*:
assumes *finite T S* ⊆ ⋃ *T card T* < *card S*
shows ∃ *x* ∈ *S*. ∃ *y* ∈ *S*. ∃ *X* ∈ *T*. *x* ≠ *y* ∧ *x* ∈ *X* ∧ *y* ∈ *X*

proof (*rule ccontr*)
assume ¬ (∃ *x* ∈ *S*. ∃ *y* ∈ *S*. ∃ *X* ∈ *T*. *x* ≠ *y* ∧ *x* ∈ *X* ∧ *y* ∈ *X*)
hence ∗: ∀ *X* ∈ *T*. *card* (*S* ∩ *X*) ≤ *1*
using *card-Int-if-either-in* **by** *metis*
have *card T* < *card* (*S* ∩ ⋃ *T*)
using *Int-absorb2 assms* **by** *fastforce*
also have ... ≤ (∑ *X* ∈ *T*. *card* (*S* ∩ *X*))
using *assms*(*1*) *card-Int-Un-le-Sum-card-Int* **by** *blast*
also have ... ≤ *card T*
using ∗ *sum-mono* **by** *fastforce*
finally show *False* **by** *simp*
qed

1.6.4 Delta Sparse Points within a Square

lemma *max-points-square*:
assumes ∀ *p* ∈ *ps*. *p* ∈ *cbox* (*x*, *y*) (*x* + *δ*, *y* + *δ*) *sparse δ ps 0* ≤ *δ*
shows *card ps* ≤ *4*
proof (*cases δ* = *0*)
case *True*
hence { (*x*, *y*) } = *cbox* (*x*, *y*) (*x* + *δ*, *y* + *δ*)
using *cbox-def* **by** *simp*
hence ∀ *p* ∈ *ps*. *p* = (*x*, *y*)
using *assms*(*1*) **by** *blast*
hence ∀ *p* ∈ *ps*. ∀ *q* ∈ *ps*. *p* = *q*
apply (*auto split*: *prod.splits*) **by** (*metis of-int-eq-iff*)+
thus *?thesis*
using *card-le-1-if-pairwise-eq* **by** *force*
next
case *False*
hence *δ*: *0* < *δ*
using *assms*(*3*) **by** *simp*
show *?thesis*
proof (*rule ccontr*)
assume *A*: ¬ (*card ps* ≤ *4*)
define *PS* **where** *PS-def*: *PS* = (*λ*(*x*, *y*). (*real-of-int x*, *real-of-int y*)) ‘ *ps*
have *inj-on* (*λ*(*x*, *y*). (*real-of-int x*, *real-of-int y*)) *ps*
using *inj-on-def* **by** *fastforce*
hence ∗: ¬ (*card PS* ≤ *4*)
using *A PS-def* **by** (*simp add*: *card-image*)

let *?x′* = *x* + *δ* / *2*
let *?y′* = *y* + *δ* / *2*

let *?ll* = *cbox* (*x* , *y*) (*?x′* , *?y′*)
let *?lu* = *cbox* (*x* , *?y′*) (*?x′* , *y* + *δ*)
let *?rl* = *cbox* (*?x′*, *y*) (*x* + *δ*, *?y′*)
let *?ru* = *cbox* (*?x′*, *?y′*) (*x* + *δ*, *y* + *δ*)

let *?sq* = { *?ll*, *?lu*, *?rl*, *?ru* }

have *card-le-4*: *card ?sq* ≤ *4*
by (*simp add*: *card-insert-le-m1*)

have *cbox* (*x*, *y*) (*?x′*, *y* + *δ*) = *?ll* ∪ *?lu*
using *cbox-top-un assms*(*3*) **by** *auto*
moreover have *cbox* (*?x′*, *y*) (*x* + *δ*, *y* + *δ*) = *?rl* ∪ *?ru*
using *cbox-top-un assms*(*3*) **by** *auto*
moreover have *cbox* (*x*, *y*) (*?x′*, *y* + *δ*) ∪ *cbox* (*?x′*, *y*) (*x* + *δ*, *y* + *δ*) = *cbox* (*x*, *y*) (*x* + *δ*, *y* + *δ*)
using *cbox-right-un assms*(*3*) **by** *simp*
ultimately have *?ll* ∪ *?lu* ∪ *?rl* ∪ *?ru* = *cbox* (*x*, *y*) (*x* + *δ*, *y* + *δ*)
by *blast*

hence *PS* ⊆ ⋃(*?sq*)
using *assms*(*1*) *PS-def* **by** (*auto split*: *prod.splits*)
moreover have *card ?sq* < *card PS*
using ∗ *card-insert-le-m1 card-le-4* **by** *linarith*
moreover have *finite ?sq*
by *simp*
ultimately have ∃ *p*₀ ∈ *PS*. ∃ *p*₁ ∈ *PS*. ∃ *S* ∈ *?sq*. (*p*₀ ≠ *p*₁ ∧ *p*₀ ∈ *S* ∧ *p*₁ ∈ *S*)
using *pigeonhole*[*of ?sq PS*] **by** *blast*
then obtain *S p*₀ *p*₁ **where** #: *p*₀ ∈ *PS p*₁ ∈ *PS S* ∈ *?sq p*₀ ≠ *p*₁ *p*₀ ∈ *S p*₁ ∈ *S*
by *blast*

have *D*: *0* ≤ *δ* / *2*
using *assms*(*3*) **by** *simp*
have *LL*: ∀ *p*₀ ∈ *?ll*. ∀ *p*₁ ∈ *?ll*. *dist p*₀ *p*₁ ≤ *sqrt 2* ∗ (*δ* / *2*)
using *cbox-max-dist*[*of* (*x*, *y*) *x y* (*?x′*, *?y′*) *δ* / *2*] *D* **by** *auto*
have *LU*: ∀ *p*₀ ∈ *?lu*. ∀ *p*₁ ∈ *?lu*. *dist p*₀ *p*₁ ≤ *sqrt 2* ∗ (*δ* / *2*)
using *cbox-max-dist*[*of* (*x*, *?y′*) *x ?y′* (*?x′*, *y* + *δ*) *δ* / *2*] *D* **by** *auto*
have *RL*: ∀ *p*₀ ∈ *?rl*. ∀ *p*₁ ∈ *?rl*. *dist p*₀ *p*₁ ≤ *sqrt 2* ∗ (*δ* / *2*)
using *cbox-max-dist*[*of* (*?x′*, *y*) *?x′ y* (*x* + *δ*, *?y′*) *δ* / *2*] *D* **by** *auto*
have *RU*: ∀ *p*₀ ∈ *?ru*. ∀ *p*₁ ∈ *?ru*. *dist p*₀ *p*₁ ≤ *sqrt 2* ∗ (*δ* / *2*)
using *cbox-max-dist*[*of* (*?x′*, *?y′*) *?x′ ?y′* (*x* + *δ*, *y* + *δ*) *δ* / *2*] *D* **by** *auto*

have ∀ *p*₀ ∈ *S*. ∀ *p*₁ ∈ *S*. *dist p*₀ *p*₁ ≤ *sqrt 2* ∗ (*δ* / *2*)
using # *LL LU RL RU* **by** *blast*
hence *dist p*₀ *p*₁ ≤ (*sqrt 2* / *2*) ∗ *δ*
using # **by** *simp*
moreover have (*sqrt 2* / *2*) ∗ *δ* < *δ*
using *sqrt2-less-2 δ* **by** *simp*
ultimately have *dist p*₀ *p*₁ < *δ*
by *simp*
moreover have *δ* ≤ *dist p*₀ *p*₁
using *assms*(*2*) # *sparse-def PS-def* **by** *auto*

```
    ultimately show False
      by simp
  qed
qed

end
```

2 Closest Pair Algorithm

```
theory Closest-Pair
  imports Common
begin
```

Formalization of a slightly optimized divide-and-conquer algorithm solving the Closest Pair Problem based on the presentation of Cormen *et al.* [1].

2.1 Functional Correctness Proof

2.1.1 Combine Step

```
fun find-closest-tm :: point ⇒ real ⇒ point list ⇒ point tm where
  find-closest-tm - - [] =1 return undefined
| find-closest-tm - - [p] =1 return p
| find-closest-tm p δ (p0 # ps) =1 (
    if δ ≤ snd p0 − snd p then
      return p0
    else
      do {
        p1 <− find-closest-tm p (min δ (dist p p0)) ps;
        if dist p p0 ≤ dist p p1 then
          return p0
        else
          return p1
      }
  )

fun find-closest :: point ⇒ real ⇒ point list ⇒ point where
  find-closest - - [] = undefined
| find-closest - - [p] = p
| find-closest p δ (p0 # ps) = (
    if δ ≤ snd p0 − snd p then
      p0
    else
      let p1 = find-closest p (min δ (dist p p0)) ps in
      if dist p p0 ≤ dist p p1 then
        p0
      else
        p1
```

```
)

lemma find-closest-eq-val-find-closest-tm:
  val (find-closest-tm p δ ps) = find-closest p δ ps
  by (induction p δ ps rule: find-closest.induct) (auto simp: Let-def)

lemma find-closest-set:
  0 < length ps ⟹ find-closest p δ ps ∈ set ps
  by (induction p δ ps rule: find-closest.induct)
    (auto simp: Let-def)

lemma find-closest-dist:
  assumes sorted-snd (p # ps) ∃ q ∈ set ps. dist p q < δ
  shows ∀ q ∈ set ps. dist p (find-closest p δ ps) ≤ dist p q
  using assms
proof (induction p δ ps rule: find-closest.induct)
  case (3 p δ p₀ p₂ ps)
  let ?ps = p₀ # p₂ # ps
  define p₁ where p₁-def: p₁ = find-closest p (min δ (dist p p₀)) (p₂ # ps)
  have A: ¬ δ ≤ snd p₀ − snd p
  proof (rule ccontr)
    assume B: ¬ ¬ δ ≤ snd p₀ − snd p
    have ∀ q ∈ set ?ps. snd p ≤ snd q
      using sorted-snd-def 3.prems(1) by simp
    moreover have ∀ q ∈ set ?ps. δ ≤ snd q − snd p
      using sorted-snd-def 3.prems(1) B by auto
    ultimately have ∀ q ∈ set ?ps. δ ≤ dist (snd p) (snd q)
      using dist-real-def by simp
    hence ∀ q ∈ set ?ps. δ ≤ dist p q
      using dist-snd-le order-trans
      apply (auto split: prod.splits) by fastforce+
    thus False
      using 3.prems(2) by fastforce
  qed
  show ?case
  proof cases
    assume ∃ q ∈ set (p₂ # ps). dist p q < min δ (dist p p₀)
    hence ∀ q ∈ set (p₂ # ps). dist p p₁ ≤ dist p q
      using 3.IH 3.prems(1) A p₁-def sorted-snd-def by simp
    thus ?thesis
      using p₁-def A by (auto split: prod.splits)
  next
    assume B: ¬ (∃ q ∈ set (p₂ # ps). dist p q < min δ (dist p p₀))
    hence dist p p₀ < δ
      using 3.prems(2) p₁-def by auto
    hence C: ∀ q ∈ set ?ps. dist p p₀ ≤ dist p q
      using p₁-def B by auto
    have p₁ ∈ set (p₂ # ps)
      using p₁-def find-closest-set by blast
```

```
    hence dist p p₀ ≤ dist p p₁
      using p₁-def C by auto
    thus ?thesis
      using p₁-def A C by (auto split: prod.splits)
  qed
qed auto

declare find-closest.simps [simp del]

fun find-closest-pair-tm :: (point * point) ⇒ point list ⇒ (point × point) tm where
  find-closest-pair-tm (c₀, c₁) [] =1 return (c₀, c₁)
| find-closest-pair-tm (c₀, c₁) [-] =1 return (c₀, c₁)
| find-closest-pair-tm (c₀, c₁) (p₀ # ps) =1 (
    do {
      p₁ <− find-closest-tm p₀ (dist c₀ c₁) ps;
      if dist c₀ c₁ ≤ dist p₀ p₁ then
        find-closest-pair-tm (c₀, c₁) ps
      else
        find-closest-pair-tm (p₀, p₁) ps
    }
  )

fun find-closest-pair :: (point * point) ⇒ point list ⇒ (point × point) where
  find-closest-pair (c₀, c₁) [] = (c₀, c₁)
| find-closest-pair (c₀, c₁) [-] = (c₀, c₁)
| find-closest-pair (c₀, c₁) (p₀ # ps) = (
    let p₁ = find-closest p₀ (dist c₀ c₁) ps in
    if dist c₀ c₁ ≤ dist p₀ p₁ then
      find-closest-pair (c₀, c₁) ps
    else
      find-closest-pair (p₀, p₁) ps
  )

lemma find-closest-pair-eq-val-find-closest-pair-tm:
  val (find-closest-pair-tm (c₀, c₁) ps) = find-closest-pair (c₀, c₁) ps
  by (induction (c₀, c₁) ps arbitrary: c₀ c₁ rule: find-closest-pair.induct)
     (auto simp: Let-def find-closest-eq-val-find-closest-tm)

lemma find-closest-pair-set:
  assumes (C₀, C₁) = find-closest-pair (c₀, c₁) ps
  shows (C₀ ∈ set ps ∧ C₁ ∈ set ps) ∨ (C₀ = c₀ ∧ C₁ = c₁)
  using assms
proof (induction (c₀, c₁) ps arbitrary: c₀ c₁ C₀ C₁ rule: find-closest-pair.induct)
  case (3 c₀ c₁ p₀ p₂ ps)
  define p₁ where p₁-def: p₁ = find-closest p₀ (dist c₀ c₁) (p₂ # ps)
  hence A: p₁ ∈ set (p₂ # ps)
    using find-closest-set by blast
  show ?case
  proof (cases dist c₀ c₁ ≤ dist p₀ p₁)
```

```
    case True
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (c0, c1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence (C0' ∈ set (p2 # ps) ∧ C1' ∈ set (p2 # ps)) ∨ (C0' = c0 ∧ C1' = c1)
      using 3.hyps(1) True p1-def by blast
    moreover have C0 = C0' C1 = C1'
      using defs True 3.prems by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
      by auto
  next
    case False
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (p0, p1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence (C0' ∈ set (p2 # ps) ∧ C1' ∈ set (p2 # ps)) ∨ (C0' = p0 ∧ C1' = p1)
      using 3.hyps(2) p1-def False by blast
    moreover have C0 = C0' C1 = C1'
      using defs False 3.prems by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
      using A by auto
  qed
qed auto

lemma find-closest-pair-c0-ne-c1:
  c0 ≠ c1 ⟹ distinct ps ⟹ (C0, C1) = find-closest-pair (c0, c1) ps ⟹ C0 ≠
C1
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case (3 c0 c1 p0 p2 ps)
  define p1 where p1-def: p1 = find-closest p0 (dist c0 c1) (p2 # ps)
  hence A: p0 ≠ p1
    using 3.prems(1,2)
   by (metis distinct.simps(2) find-closest-set length-pos-if-in-set list.set-intros(1))
  show ?case
  proof (cases dist c0 c1 ≤ dist p0 p1)
    case True
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (c0, c1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence C0' ≠ C1'
      using 3.hyps(1) 3.prems(1,2) True p1-def by simp
    moreover have C0 = C0' C1 = C1'
      using defs True 3.prems(3) by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
      by simp
  next
```

```
    case False
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (p0, p1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence C0' ≠ C1'
      using 3.hyps(2) 3.prems(2) A False p1-def by simp
    moreover have C0 = C0' C1 = C1'
      using defs False 3.prems(3) by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
      by simp
  qed
qed auto

lemma find-closest-pair-dist-mono:
  assumes (C0, C1) = find-closest-pair (c0, c1) ps
  shows dist C0 C1 ≤ dist c0 c1
  using assms
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case (3 c0 c1 p0 p2 ps)
  define p1 where p1-def: p1 = find-closest p0 (dist c0 c1) (p2 # ps)
  show ?case
  proof (cases dist c0 c1 ≤ dist p0 p1)
    case True
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (c0, c1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence dist C0' C1' ≤ dist c0 c1
      using 3.hyps(1) True p1-def by simp
    moreover have C0 = C0' C1 = C1'
      using defs True 3.prems by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
      by simp
  next
    case False
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (p0, p1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence dist C0' C1' ≤ dist p0 p1
      using 3.hyps(2) False p1-def by blast
    moreover have C0 = C0' C1 = C1'
      using defs False 3.prems(1) by (auto split: prod.splits, metis Pair-inject)+
    ultimately show ?thesis
      using False by simp
  qed
qed auto
```

lemma *find-closest-pair-dist*:
assumes *sorted-snd ps* $(C_0, C_1) =$ *find-closest-pair* (c_0, c_1) *ps*
shows *sparse* $(dist\ C_0\ C_1)$ $(set\ ps)$
using *assms*
proof (*induction* (c_0, c_1) *ps arbitrary*: $c_0\ c_1\ C_0\ C_1$ *rule*: *find-closest-pair.induct*)
case (*3* $c_0\ c_1\ p_0\ p_2\ ps$)
define p_1 **where** p_1*-def*: $p_1 =$ *find-closest* p_0 $(dist\ c_0\ c_1)$ $(p_2 \# ps)$
show *?case*
proof *cases*
assume $\exists p \in set\ (p_2 \# ps).\ dist\ p_0\ p < dist\ c_0\ c_1$
hence *A*: $\forall p \in set\ (p_2 \# ps).\ dist\ p_0\ p_1 \leq dist\ p_0\ p\ dist\ p_0\ p_1 < dist\ c_0\ c_1$
using p_1*-def find-closest-dist 3.prems*(*1*) *le-less-trans* **by** *blast+*
obtain $C_0'\ C_1'$ **where** C'*-def*: $(C_0', C_1') =$ *find-closest-pair* (p_0, p_1) $(p_2 \# ps)$
using *prod.collapse* **by** *blast*
hence *B*: $(C_0', C_1') =$ *find-closest-pair* (c_0, c_1) $(p_0 \# p_2 \# ps)$
using *A*(*2*) p_1*-def* **by** *simp*
have *sparse* $(dist\ C_0'\ C_1')$ $(set\ (p_2 \# ps))$
using *3.hyps*(*2*)[*of* $p_1\ C_0'\ C_1'$] p_1*-def* C'*-def 3.prems*(*1*) *A*(*2*) *sorted-snd-def* **by** *fastforce*
moreover have $dist\ C_0'\ C_1' \leq dist\ p_0\ p_1$
using C'*-def find-closest-pair-dist-mono* **by** *blast*
ultimately have *sparse* $(dist\ C_0'\ C_1')$ $(set\ (p_0 \# p_2 \# ps))$
using *A sparse-identity order-trans* **by** *blast*
thus *?thesis*
using *B* **by** (*metis 3.prems*(*2*) *Pair-inject*)
next
assume *A*: $\neg\ (\exists p \in set\ (p_2 \# ps).\ dist\ p_0\ p < dist\ c_0\ c_1)$
hence *B*: $dist\ c_0\ c_1 \leq dist\ p_0\ p_1$
using *find-closest-set*[*of* $p_2 \# ps\ p_0\ dist\ c_0\ c_1$] p_1*-def* **by** *auto*
obtain $C_0'\ C_1'$ **where** C'*-def*: $(C_0', C_1') =$ *find-closest-pair* (c_0, c_1) $(p_2 \# ps)$
using *prod.collapse* **by** *blast*
hence *C*: $(C_0', C_1') =$ *find-closest-pair* (c_0, c_1) $(p_0 \# p_2 \# ps)$
using *B* p_1*-def* **by** *simp*
have *sparse* $(dist\ C_0'\ C_1')$ $(set\ (p_2 \# ps))$
using *3.hyps*(*1*)[*of* $p_1\ C_0'\ C_1'$] p_1*-def* C'*-def B 3.prems sorted-snd-def* **by** *simp*
moreover have $dist\ C_0'\ C_1' \leq dist\ c_0\ c_1$
using C'*-def find-closest-pair-dist-mono* **by** *blast*
ultimately have *sparse* $(dist\ C_0'\ C_1')$ $(set\ (p_0 \# p_2 \# ps))$
using *A sparse-identity*[*of* $dist\ C_0'\ C_1'\ p_2 \# ps\ p_0$] *order-trans* **by** *force*
thus *?thesis*
using *C* **by** (*metis 3.prems*(*2*) *Pair-inject*)
qed
qed (*auto simp*: *sparse-def*)

declare *find-closest-pair.simps* [*simp del*]

fun *combine-tm* :: (*point* × *point*) ⇒ (*point* × *point*) ⇒ *int* ⇒ *point list* ⇒ (*point* × *point*) *tm* **where**
  combine-tm (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps* =*1* (
   let (c_0, c_1) = *if dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}) *in*
   do {
     ps′ <− *filter-tm* (λ*p*. *dist p* (*l*, *snd p*) < *dist* c_0 c_1) *ps*;
     find-closest-pair-tm (c_0, c_1) *ps′*
   }
)

fun *combine* :: (*point* × *point*) ⇒ (*point* × *point*) ⇒ *int* ⇒ *point list* ⇒ (*point* × *point*) **where**
  combine (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps* = (
   let (c_0, c_1) = *if dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}) *in*
   let ps′ = *filter* (λ*p*. *dist p* (*l*, *snd p*) < *dist* c_0 c_1) *ps in*
   find-closest-pair (c_0, c_1) *ps′*
)

lemma *combine-eq-val-combine-tm*:
  val (*combine-tm* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
  by (*auto simp*: *filter-eq-val-filter-tm find-closest-pair-eq-val-find-closest-pair-tm*)

lemma *combine-set*:
  assumes (c_0, c_1) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
  shows (c_0 ∈ *set ps* ∧ c_1 ∈ *set ps*) ∨ (c_0 = p_{0L} ∧ c_1 = p_{1L}) ∨ (c_0 = p_{0R} ∧ c_1 = p_{1R})
proof −
  obtain $C_0{}'$ $C_1{}'$ **where** *C′-def*: ($C_0{}'$, $C_1{}'$) = (*if dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}))
    by *metis*
  define *ps′* **where** *ps′-def*: *ps′* = *filter* (λ*p*. *dist p* (*l*, *snd p*) < *dist* $C_0{}'$ $C_1{}'$) *ps*
  obtain C_0 C_1 **where** *C-def*: (C_0, C_1) = *find-closest-pair* ($C_0{}'$, $C_1{}'$) *ps′*
    using *prod.collapse* **by** *blast*
  note *defs* = *C′-def ps′-def C-def*
  have (C_0 ∈ *set ps′* ∧ C_1 ∈ *set ps′*) ∨ (C_0 = $C_0{}'$ ∧ C_1 = $C_1{}'$)
    using *C-def find-closest-pair-set* **by** *blast*+
  hence (C_0 ∈ *set ps* ∧ C_1 ∈ *set ps*)∨ (C_0 = $C_0{}'$ ∧ C_1 = $C_1{}'$)
    using *ps′-def* **by** *auto*
  moreover have C_0 = c_0 C_1 = c_1
   using *assms defs* **apply** (*auto split*: *if-splits prod.splits*) **by** (*metis Pair-inject*)+
  ultimately show *?thesis*
    using *C′-def* **by** (*auto split*: *if-splits*)
qed

lemma *combine-c0-ne-c1*:
  assumes p_{0L} ≠ p_{1L} p_{0R} ≠ p_{1R} *distinct ps*
  assumes (c_0, c_1) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
  shows c_0 ≠ c_1

proof −
obtain C_0' C_1' **where** *C'-def*: $(C_0', C_1') = (\textit{if } \textit{dist } p_{0L}\ p_{1L} < \textit{dist } p_{0R}\ p_{1R}\ \textit{then } (p_{0L}, p_{1L})\ \textit{else } (p_{0R}, p_{1R}))$
by *metis*
define ps' **where** *ps'-def*: $ps' = \textit{filter } (\lambda p.\ \textit{dist } p\ (l, \textit{snd } p) < \textit{dist } C_0'\ C_1')\ ps$
obtain C_0 C_1 **where** *C-def*: $(C_0, C_1) = \textit{find-closest-pair } (C_0', C_1')\ ps'$
using *prod.collapse* **by** *blast*
note *defs* = *C'-def ps'-def C-def*
have $C_0 \neq C_1$
using *defs find-closest-pair-c0-ne-c1*[*of* C_0' C_1' ps'] *assms* **by** (*auto split*: *if-splits*)
moreover have $C_0 = c_0$ $C_1 = c_1$
using *assms*(*4*) *defs* **apply** (*auto split*: *if-splits prod.splits*) **by** (*metis Pair-inject*)+
ultimately show *?thesis*
by *blast*
qed

lemma *combine-dist*:
assumes *sorted-snd ps set ps* = $ps_L \cup ps_R$
assumes $\forall p \in ps_L.\ \textit{fst } p \leq l\ \forall p \in ps_R.\ l \leq \textit{fst } p$
assumes *sparse* (*dist* $p_{0L}\ p_{1L}$) ps_L *sparse* (*dist* $p_{0R}\ p_{1R}$) ps_R
assumes $(c_0, c_1) = \textit{combine } (p_{0L}, p_{1L})\ (p_{0R}, p_{1R})\ l\ ps$
shows *sparse* (*dist* $c_0\ c_1$) (*set ps*)
proof −
obtain C_0' C_1' **where** *C'-def*: $(C_0', C_1') = (\textit{if } \textit{dist } p_{0L}\ p_{1L} < \textit{dist } p_{0R}\ p_{1R}\ \textit{then } (p_{0L}, p_{1L})\ \textit{else } (p_{0R}, p_{1R}))$
by *metis*
define ps' **where** *ps'-def*: $ps' = \textit{filter } (\lambda p.\ \textit{dist } p\ (l, \textit{snd } p) < \textit{dist } C_0'\ C_1')\ ps$
obtain C_0 C_1 **where** *C-def*: $(C_0, C_1) = \textit{find-closest-pair } (C_0', C_1')\ ps'$
using *prod.collapse* **by** *blast*
note *defs* = *C'-def ps'-def C-def*
have *EQ*: $C_0 = c_0$ $C_1 = c_1$
using *defs assms*(*7*) **apply** (*auto split*: *if-splits prod.splits*) **by** (*metis Pair-inject*)+
have *ps'*: $ps' = \textit{filter } (\lambda p.\ l - \textit{dist } C_0'\ C_1' < \textit{fst } p \wedge \textit{fst } p < l + \textit{dist } C_0'\ C_1')\ ps$
using *ps'-def dist-transform* **by** *simp*
have ps_L: *sparse* (*dist* $C_0'\ C_1'$) ps_L
using *assms*(*3*,*5*) *C'-def sparse-def* **apply** (*auto split*: *if-splits*) **by** *force*+
have ps_R: *sparse* (*dist* $C_0'\ C_1'$) ps_R
using *assms*(*4*,*6*) *C'-def sparse-def* **apply** (*auto split*: *if-splits*) **by** *force*+
have *sorted-snd ps'*
using *ps'-def assms*(*1*) *sorted-snd-def sorted-wrt-filter* **by** *blast*
hence ∗: *sparse* (*dist* $C_0\ C_1$) (*set ps'*)
using *find-closest-pair-dist C-def* **by** *simp*
have $\forall p_0 \in \textit{set } ps.\ \forall p_1 \in \textit{set } ps.\ p_0 \neq p_1 \wedge \textit{dist } p_0\ p_1 < \textit{dist } C_0'\ C_1' \longrightarrow p_0 \in \textit{set } ps' \wedge p_1 \in \textit{set } ps'$
using *set-band-filter ps' ps*$_L$ *ps*$_R$ *assms*(*2*,*3*,*4*) **by** *blast*
moreover have *dist* $C_0\ C_1 \leq \textit{dist } C_0'\ C_1'$
using *C-def find-closest-pair-dist-mono* **by** *blast*

ultimately have $\forall p_0 \in set\ ps.\ \forall p_1 \in set\ ps.\ p_0 \neq p_1 \wedge dist\ p_0\ p_1 < dist\ C_0\ C_1 \longrightarrow p_0 \in set\ ps' \wedge p_1 \in set\ ps'$
by *simp*
hence *sparse* (*dist* C_0 C_1) (*set ps*)
using *sparse-def* $*$ **by** (*meson not-less*)
thus *?thesis*
using *EQ* **by** *blast*
qed

declare *combine.simps* [*simp del*]
declare *combine-tm.simps*[*simp del*]

2.1.2 Divide and Conquer Algorithm

declare *split-at-take-drop-conv* [*simp add*]

function *closest-pair-rec-tm* :: *point list* ⇒ (*point list* × *point* × *point*) *tm* **where**
closest-pair-rec-tm xs =1 (
do {
n <− *length-tm xs*;
if n ≤ *3 then*
do {
ys <− *mergesort-tm snd xs*;
p <− *closest-pair-bf-tm xs*;
return (*ys*, *p*)
}
else
do {
(xs_L, xs_R) <− *split-at-tm* (*n div 2*) *xs*;
(ys_L, p_{0L}, p_{1L}) <− *closest-pair-rec-tm* xs_L;
(ys_R, p_{0R}, p_{1R}) <− *closest-pair-rec-tm* xs_R;
ys <− *merge-tm snd* ys_L ys_R;
(p_0, p_1) <− *combine-tm* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) (*fst* (*hd* xs_R)) *ys*;
return (*ys*, p_0, p_1)
}
}
)
by *pat-completeness auto*
termination *closest-pair-rec-tm*
by (*relation Wellfounded.measure* (λ*xs*. *length xs*))
(*auto simp add*: *length-eq-val-length-tm split-at-eq-val-split-at-tm*)

function *closest-pair-rec* :: *point list* ⇒ (*point list* $*$ *point* $*$ *point*) **where**
closest-pair-rec xs = (
let n = *length xs in*
if n ≤ *3 then*
(*mergesort snd xs*, *closest-pair-bf xs*)
else
let (xs_L, xs_R) = *split-at* (*n div 2*) *xs in*

```
    let (ys_L, p_{0L}, p_{1L}) = closest-pair-rec xs_L in
    let (ys_R, p_{0R}, p_{1R}) = closest-pair-rec xs_R in
    let ys = merge snd ys_L ys_R in
    (ys, combine (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) (fst (hd xs_R)) ys)
 )
 by pat-completeness auto
termination closest-pair-rec
 by (relation Wellfounded.measure (λxs. length xs))
   (auto simp: Let-def)

declare split-at-take-drop-conv [simp del]

lemma closest-pair-rec-simps:
 assumes n = length xs ¬ (n ≤ 3)
 shows closest-pair-rec xs = (
   let (xs_L, xs_R) = split-at (n div 2) xs in
   let (ys_L, p_{0L}, p_{1L}) = closest-pair-rec xs_L in
   let (ys_R, p_{0R}, p_{1R}) = closest-pair-rec xs_R in
   let ys = merge snd ys_L ys_R in
   (ys, combine (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) (fst (hd xs_R)) ys)
 )
 using assms by (auto simp: Let-def)

declare closest-pair-rec.simps [simp del]

lemma closest-pair-rec-eq-val-closest-pair-rec-tm:
 val (closest-pair-rec-tm xs) = closest-pair-rec xs
proof (induction rule: length-induct)
 case (1 xs)
 define n where n = length xs
 obtain xs_L xs_R where xs-def: (xs_L, xs_R) = split-at (n div 2) xs
   by (metis surj-pair)
 note defs = n-def xs-def
 show ?case
 proof cases
   assume n ≤ 3
   then show ?thesis
     using defs
     by (auto simp: length-eq-val-length-tm mergesort-eq-val-mergesort-tm
                closest-pair-bf-eq-val-closest-pair-bf-tm closest-pair-rec.simps)
 next
   assume asm: ¬ n ≤ 3
   have length xs_L < length xs length xs_R < length xs
     using asm defs by (auto simp: split-at-take-drop-conv)
   hence val (closest-pair-rec-tm xs_L) = closest-pair-rec xs_L
       val (closest-pair-rec-tm xs_R) = closest-pair-rec xs_R
     using 1.IH by blast+
   thus ?thesis
     using asm defs
```

```
    apply (subst closest-pair-rec.simps, subst closest-pair-rec-tm.simps)
    by (auto simp del: closest-pair-rec-tm.simps
          simp add: Let-def length-eq-val-length-tm merge-eq-val-merge-tm
                split-at-eq-val-split-at-tm combine-eq-val-combine-tm
          split: prod.split)
  qed
qed

lemma closest-pair-rec-set-length-sorted-snd:
  assumes (ys, p) = closest-pair-rec xs
  shows set ys = set xs ∧ length ys = length xs ∧ sorted-snd ys
  using assms
proof (induction xs arbitrary: ys p rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    thus ?thesis using 1.prems sorted-snd-def
      by (auto simp: mergesort closest-pair-rec.simps)
  next
    case False

    obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
      using prod.collapse by blast
    define L where L = fst (hd XS_R)
    obtain YS_L P_L where YSP_L-def: (YS_L, P_L) = closest-pair-rec XS_L
      using prod.collapse by blast
    obtain YS_R P_R where YSP_R-def: (YS_R, P_R) = closest-pair-rec XS_R
      using prod.collapse by blast
    define YS where YS = merge (λp. snd p) YS_L YS_R
    define P where P = combine P_L P_R L YS
    note defs = XS_LR-def L-def YSP_L-def YSP_R-def YS-def P-def

    have length XS_L < length xs length XS_R < length xs
      using False defs by (auto simp: split-at-take-drop-conv)
    hence IH: set XS_L = set YS_L set XS_R = set YS_R
          length XS_L = length YS_L length XS_R = length YS_R
          sorted-snd YS_L sorted-snd YS_R
      using 1.IH defs by metis+

    have set xs = set XS_L ∪ set XS_R
      using defs by (auto simp: set-take-drop split-at-take-drop-conv)
    hence SET: set xs = set YS
      using set-merge IH(1,2) defs by fast

    have length xs = length XS_L + length XS_R
      using defs by (auto simp: split-at-take-drop-conv)
    hence LENGTH: length xs = length YS
```

```
    using IH(3,4) length-merge defs by metis

  have SORTED: sorted-snd YS
    using IH(5,6) by (simp add: defs sorted-snd-def sorted-merge)

  have (YS, P) = closest-pair-rec xs
   using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
  hence (ys, p) = (YS, P)
    using 1.prems by argo
  thus ?thesis
    using SET LENGTH SORTED by simp
 qed
qed

lemma closest-pair-rec-distinct:
 assumes distinct xs (ys, p) = closest-pair-rec xs
 shows distinct ys
 using assms
proof (induction xs arbitrary: ys p rule: length-induct)
 case (1 xs)
 let ?n = length xs
 show ?case
 proof (cases ?n ≤ 3)
   case True
   thus ?thesis using 1.prems
     by (auto simp: mergesort closest-pair-rec.simps)
 next
   case False

   obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
     using prod.collapse by blast
   define L where L = fst (hd XS_R)
   obtain YS_L P_L where YSP_L-def: (YS_L, P_L) = closest-pair-rec XS_L
     using prod.collapse by blast
   obtain YS_R P_R where YSP_R-def: (YS_R, P_R) = closest-pair-rec XS_R
     using prod.collapse by blast
   define YS where YS = merge (λp. snd p) YS_L YS_R
   define P where P = combine P_L P_R L YS
   note defs = XS_LR-def L-def YSP_L-def YSP_R-def YS-def P-def

   have length XS_L < length xs length XS_R < length xs
     using False defs by (auto simp: split-at-take-drop-conv)
   moreover have distinct XS_L distinct XS_R
     using 1.prems(1) defs by (auto simp: split-at-take-drop-conv)
   ultimately have IH: distinct YS_L distinct YS_R
     using 1.IH defs by blast+

   have set XS_L ∩ set XS_R = {}
    using 1.prems(1) defs by (auto simp: split-at-take-drop-conv set-take-disj-set-drop-if-distinct)
```

```
    moreover have set XS_L = set YS_L set XS_R = set YS_R
      using closest-pair-rec-set-length-sorted-snd defs by blast+
    ultimately have set YS_L ∩ set YS_R = {}
      by blast
    hence DISTINCT: distinct YS
      using distinct-merge IH defs by blast

    have (YS, P) = closest-pair-rec xs
     using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
    hence (ys, p) = (YS, P)
      using 1.prems by argo
    thus ?thesis
      using DISTINCT by blast
  qed
qed

lemma closest-pair-rec-c0-c1:
  assumes 1 < length xs distinct xs (ys, c_0, c_1) = closest-pair-rec xs
  shows c_0 ∈ set xs ∧ c_1 ∈ set xs ∧ c_0 ≠ c_1
  using assms
proof (induction xs arbitrary: ys c_0 c_1 rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence (c_0, c_1) = closest-pair-bf xs
      using 1.prems(3) closest-pair-rec.simps by simp
    thus ?thesis
      using 1.prems(1,2) closest-pair-bf-c0-c1 by simp
  next
    case False

    obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
      using prod.collapse by blast
    define L where L = fst (hd XS_R)

   obtain YS_L C_0L C_1L where YSC_01L-def: (YS_L, C_0L, C_1L) = closest-pair-rec
XS_L
      using prod.collapse by metis
   obtain YS_R C_0R C_1R where YSC_01R-def: (YS_R, C_0R, C_1R) = closest-pair-rec
XS_R
      using prod.collapse by metis

    define YS where YS = merge (λp. snd p) YS_L YS_R
    obtain C_0 C_1 where C_01-def: (C_0, C_1) = combine (C_0L, C_1L) (C_0R, C_1R)
L YS
      using prod.collapse by metis
    note defs = XS_LR-def L-def YSC_01L-def YSC_01R-def YS-def C_01-def
```

```
    have 1 < length XS_L length XS_L < length xs distinct XS_L
      using False 1.prems(2) defs by (auto simp: split-at-take-drop-conv)
    hence C_0L ∈ set XS_L C_1L ∈ set XS_L and IHL1: C_0L ≠ C_1L
      using 1.IH defs by metis+
    hence IHL2: C_0L ∈ set xs C_1L ∈ set xs
      using split-at-take-drop-conv in-set-takeD fst-conv defs by metis+

    have 1 < length XS_R length XS_R < length xs distinct XS_R
      using False 1.prems(2) defs by (auto simp: split-at-take-drop-conv)
    hence C_0R ∈ set XS_R C_1R ∈ set XS_R and IHR1: C_0R ≠ C_1R
      using 1.IH defs by metis+
    hence IHR2: C_0R ∈ set xs C_1R ∈ set xs
      using split-at-take-drop-conv in-set-dropD snd-conv defs by metis+

    have *: (YS, C_0, C_1) = closest-pair-rec xs
     using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
    have YS: set xs = set YS distinct YS
     using 1.prems(2) closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct
* by blast+

    have C_0 ∈ set xs C_1 ∈ set xs
      using combine-set IHL2 IHR2 YS defs by blast+
    moreover have C_0 ≠ C_1
      using combine-c0-ne-c1 IHL1(1) IHR1(1) YS defs by blast
    ultimately show ?thesis
      using 1.prems(3) * by (metis Pair-inject)
  qed
qed

lemma closest-pair-rec-dist:
  assumes 1 < length xs sorted-fst xs (ys, c_0, c_1) = closest-pair-rec xs
  shows sparse (dist c_0 c_1) (set xs)
  using assms
proof (induction xs arbitrary: ys c_0 c_1 rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence (c_0, c_1) = closest-pair-bf xs
      using 1.prems(3) closest-pair-rec.simps by simp
    thus ?thesis
      using 1.prems(1,3) closest-pair-bf-dist by metis
  next
    case False

    obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
      using prod.collapse by blast
```

define L **where** $L = fst\ (hd\ XS_R)$

obtain $YS_L\ C_{0L}\ C_{1L}$ **where** YSC_{01L}*-def*: $(YS_L,\ C_{0L},\ C_{1L}) = closest\text{-}pair\text{-}rec\ XS_L$
using *prod.collapse* **by** *metis*
obtain $YS_R\ C_{0R}\ C_{1R}$ **where** YSC_{01R}*-def*: $(YS_R,\ C_{0R},\ C_{1R}) = closest\text{-}pair\text{-}rec\ XS_R$
using *prod.collapse* **by** *metis*

define YS **where** $YS = merge\ (\lambda p.\ snd\ p)\ YS_L\ YS_R$
obtain $C_0\ C_1$ **where** C_{01}*-def*: $(C_0,\ C_1) = combine\ (C_{0L},\ C_{1L})\ (C_{0R},\ C_{1R})\ L\ YS$
using *prod.collapse* **by** *metis*
note *defs* $= XS_{LR}$*-def L-def* YSC_{01L}*-def* YSC_{01R}*-def YS-def* C_{01}*-def*

have *XSLR*: $XS_L = take\ (?n\ div\ 2)\ xs\ XS_R = drop\ (?n\ div\ 2)\ xs$
using *defs* **by** (*auto simp*: *split-at-take-drop-conv*)

have $1 < length\ XS_L\ length\ XS_L < length\ xs$
using *False XSLR* **by** *simp-all*
moreover have *sorted-fst* XS_L
using *1.prems*(2) *XSLR* **by** (*auto simp*: *sorted-fst-def sorted-wrt-take*)
ultimately have *L*: *sparse* $(dist\ C_{0L}\ C_{1L})\ (set\ XS_L)$
$set\ XS_L = set\ YS_L$
using *1 closest-pair-rec-set-length-sorted-snd closest-pair-rec-c0-c1* YSC_{01L}*-def* **by** *blast+*
hence *IHL*: *sparse* $(dist\ C_{0L}\ C_{1L})\ (set\ YS_L)$
by *argo*

have $1 < length\ XS_R\ length\ XS_R < length\ xs$
using *False XSLR* **by** *simp-all*
moreover have *sorted-fst* XS_R
using *1.prems*(2) *XSLR* **by** (*auto simp*: *sorted-fst-def sorted-wrt-drop*)
ultimately have *R*: *sparse* $(dist\ C_{0R}\ C_{1R})\ (set\ XS_R)$
$set\ XS_R = set\ YS_R$
using *1 closest-pair-rec-set-length-sorted-snd closest-pair-rec-c0-c1* YSC_{01R}*-def* **by** *blast+*
hence *IHR*: *sparse* $(dist\ C_{0R}\ C_{1R})\ (set\ YS_R)$
by *argo*

have $*$: $(YS,\ C_0,\ C_1) = closest\text{-}pair\text{-}rec\ xs$
using *False closest-pair-rec-simps defs* **by** (*auto simp*: *Let-def split*: *prod.split*)

have *set xs* $=$ *set YS sorted-snd YS*
using *1.prems*(2) *closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct* $*$ **by** *blast+*
moreover have $\forall\ p \in set\ YS_L.\ fst\ p \leq L$
using *False 1.prems*(2) *XSLR L-def L*(2) *sorted-fst-take-less-hd-drop* **by** *simp*
moreover have $\forall\ p \in set\ YS_R.\ L \leq fst\ p$

```
    using False 1.prems(2) XSLR L-def R(2) sorted-fst-hd-drop-less-drop by simp
   moreover have set YS = set YS_L ∪ set YS_R
    using set-merge defs by fast
   moreover have (C_0, C_1) = combine (C_0L, C_1L) (C_0R, C_1R) L YS
    by (auto simp add: defs)
   ultimately have sparse (dist C_0 C_1) (set xs)
    using combine-dist IHL IHR by auto
   moreover have (YS, C_0, C_1) = (ys, c_0, c_1)
    using 1.prems(3) * by simp
   ultimately show ?thesis
    by blast
  qed
qed

fun closest-pair-tm :: point list ⇒ (point * point) tm where
  closest-pair-tm [] =1 return undefined
| closest-pair-tm [-] =1 return undefined
| closest-pair-tm ps =1 (
    do {
      xs <- mergesort-tm fst ps;
      (-, p) <- closest-pair-rec-tm xs;
      return p
    }
  )

fun closest-pair :: point list ⇒ (point * point) where
  closest-pair [] = undefined
| closest-pair [-] = undefined
| closest-pair ps = (let (-, p) = closest-pair-rec (mergesort fst ps) in p)

lemma closest-pair-eq-val-closest-pair-tm:
  val (closest-pair-tm ps) = closest-pair ps
  by (induction ps rule: induct-list012)
     (auto simp del: closest-pair-rec-tm.simps mergesort-tm.simps
       simp add: closest-pair-rec-eq-val-closest-pair-rec-tm mergesort-eq-val-mergesort-tm
         split: prod.split)

lemma closest-pair-simps:
  1 < length ps ⟹ closest-pair ps = (let (-, p) = closest-pair-rec (mergesort fst
ps) in p)
  by (induction ps rule: induct-list012) auto

declare closest-pair.simps [simp del]

theorem closest-pair-c0-c1:
  assumes 1 < length ps distinct ps (c_0, c_1) = closest-pair ps
  shows c_0 ∈ set ps c_1 ∈ set ps c_0 ≠ c_1
  using assms closest-pair-rec-c0-c1[of mergesort fst ps]
  by (auto simp: closest-pair-simps mergesort split: prod.splits)
```

theorem *closest-pair-dist*:
assumes *1 < length ps* (c_0, c_1) = *closest-pair ps*
shows *sparse* (*dist* c_0 c_1) (*set ps*)
using *assms sorted-fst-def closest-pair-rec-dist*[*of mergesort fst ps*] *closest-pair-rec-c0-c1*[*of mergesort fst ps*]
by (*auto simp*: *closest-pair-simps mergesort split*: *prod.splits*)

2.2 Time Complexity Proof

2.2.1 Core Argument

lemma *core-argument*:
fixes δ :: *real* **and** *p* :: *point* **and** *ps* :: *point list*
assumes *distinct* (*p* # *ps*) *sorted-snd* (*p* # *ps*) *0* ≤ δ *set* (*p* # *ps*) = $ps_L \cup ps_R$
assumes ∀ *q* ∈ *set* (*p* # *ps*). *l* − δ < *fst q* ∧ *fst q* < *l* + δ
assumes ∀ *q* ∈ ps_L. *fst q* ≤ *l* ∀ *q* ∈ ps_R. *l* ≤ *fst q*
assumes *sparse* δ ps_L *sparse* δ ps_R
shows *length* (*filter* (λ*q*. *snd q* − *snd p* ≤ δ) *ps*) ≤ *7*
proof −
define *PS* **where** *PS* = *p* # *ps*
define *R* **where** *R* = *cbox* (*l* − δ, *snd p*) (*l* + δ, *snd p* + δ)
define *RPS* **where** *RPS* = { *p* ∈ *set PS*. *p* ∈ *R* }
define *LSQ* **where** *LSQ* = *cbox* (*l* − δ, *snd p*) (*l*, *snd p* + δ)
define *LSQPS* **where** *LSQPS* = { *p* ∈ ps_L. *p* ∈ *LSQ* }
define *RSQ* **where** *RSQ* = *cbox* (*l*, *snd p*) (*l* + δ, *snd p* + δ)
define *RSQPS* **where** *RSQPS* = { *p* ∈ ps_R. *p* ∈ *RSQ* }
note *defs* = *PS-def R-def RPS-def LSQ-def LSQPS-def RSQ-def RSQPS-def*

have *R* = *LSQ* ∪ *RSQ*
using *defs cbox-right-un* **by** *auto*
moreover have ∀ *p* ∈ ps_L. *p* ∈ *RSQ* ⟶ *p* ∈ *LSQ*
using *RSQ-def LSQ-def assms*(*6*) **by** *auto*
moreover have ∀ *p* ∈ ps_R. *p* ∈ *LSQ* ⟶ *p* ∈ *RSQ*
using *RSQ-def LSQ-def assms*(*7*) **by** *auto*
ultimately have *RPS* = *LSQPS* ∪ *RSQPS*
using *LSQPS-def RSQPS-def PS-def RPS-def assms*(*4*) **by** *blast*

have *sparse* δ *LSQPS*
using *assms*(*8*) *LSQPS-def sparse-def* **by** *simp*
hence *CLSQPS*: *card LSQPS* ≤ *4*
using *max-points-square*[*of LSQPS l* − δ *snd p* δ] *assms*(*3*) *LSQ-def LSQPS-def*
by *auto*

have *sparse* δ *RSQPS*
using *assms*(*9*) *RSQPS-def sparse-def* **by** *simp*
hence *CRSQPS*: *card RSQPS* ≤ *4*
using *max-points-square*[*of RSQPS l snd p* δ] *assms*(*3*) *RSQ-def RSQPS-def*
by *auto*

```
  have CRPS: card RPS ≤ 8
    using CLSQPS CRSQPS card-Un-le[of LSQPS RSQPS] ‹RPS = LSQPS ∪
RSQPS› by auto

  have set (p # filter (λq. snd q − snd p ≤ δ) ps) ⊆ RPS
  proof standard
    fix q
    assume ∗: q ∈ set (p # filter (λq. snd q − snd p ≤ δ) ps)
    hence CPS: q ∈ set PS
      using PS-def by auto
    hence snd p ≤ snd q snd q ≤ snd p + δ
      using assms(2,3) PS-def sorted-snd-def ∗ by (auto split: if-splits)
    moreover have l − δ < fst q fst q < l + δ
      using CPS assms(5) PS-def by blast+
    ultimately have q ∈ R
      using R-def mem-cbox-2D[of l − δ fst q l + δ snd p snd q snd p + δ]
      by (simp add: prod.case-eq-if)
    thus q ∈ RPS
      using CPS RPS-def by simp
  qed
  moreover have finite RPS
    by (simp add: RPS-def)
  ultimately have card (set (p # filter (λq. snd q − snd p ≤ δ) ps)) ≤ 8
    using CRPS card-mono[of RPS set (p # filter (λq. snd q − snd p ≤ δ) ps)]
by simp
  moreover have distinct (p # filter (λq. snd q − snd p ≤ δ) ps)
    using assms(1) by simp
  ultimately have length (p # filter (λq. snd q − snd p ≤ δ) ps) ≤ 8
    using assms(1) PS-def distinct-card by metis
  thus ?thesis
    by simp
qed
```

2.2.2 Combine Step

```
fun t-find-closest :: point ⇒ real ⇒ point list ⇒ nat where
  t-find-closest - - [] = 1
| t-find-closest - - [-] = 1
| t-find-closest p δ (p0 # ps) = 1 + (
    if δ ≤ snd p0 − snd p then 0
    else t-find-closest p (min δ (dist p p0)) ps
  )

lemma t-find-closest-eq-time-find-closest-tm:
  t-find-closest p δ ps = time (find-closest-tm p δ ps)
  by (induction p δ ps rule: t-find-closest.induct)
     (auto simp: time-simps)

lemma t-find-closest-mono:
```

```
  δ' ≤ δ ⟹ t-find-closest p δ' ps ≤ t-find-closest p δ ps
  by (induction rule: t-find-closest.induct)
     (auto simp: Let-def min-def)

lemma t-find-closest-cnt:
  t-find-closest p δ ps ≤ 1 + length (filter (λq. snd q − snd p ≤ δ) ps)
proof (induction p δ ps rule: t-find-closest.induct)
  case (3 p δ p₀ p₂ ps)
  show ?case
  proof (cases δ ≤ snd p₀ − snd p)
    case True
    thus ?thesis
      by simp
  next
    case False
    hence ∗: snd p₀ − snd p ≤ δ
      by simp
    have t-find-closest p δ (p₀ # p₂ # ps) = 1 + t-find-closest p (min δ (dist p
p₀)) (p₂ # ps)
      using False by simp
    also have ... ≤ 1 + 1 + length (filter (λq. snd q − snd p ≤ min δ (dist p p₀))
(p₂ # ps))
      using False 3 by simp
    also have ... ≤ 1 + 1 + length (filter (λq. snd q − snd p ≤ δ) (p₂ # ps))
      using ∗ by (meson add-le-cancel-left length-filter-P-impl-Q min.bounded-iff)
    also have ... ≤ 1 + length (filter (λq. snd q − snd p ≤ δ) (p₀ # p₂ # ps))
      using False by simp
    ultimately show ?thesis
      by simp
  qed
qed auto

corollary t-find-closest-bound:
  fixes δ :: real and p :: point and ps :: point list and l :: int
  assumes distinct (p # ps) sorted-snd (p # ps) 0 ≤ δ set (p # ps) = ps_L ∪ ps_R
  assumes ∀ p' ∈ set (p # ps). l − δ < fst p' ∧ fst p' < l + δ
  assumes ∀ p ∈ ps_L. fst p ≤ l ∀ p ∈ ps_R. l ≤ fst p
  assumes sparse δ ps_L sparse δ ps_R
  shows t-find-closest p δ ps ≤ 8
  using assms core-argument[of p ps δ ps_L ps_R l] t-find-closest-cnt[of p δ ps] by
linarith

fun t-find-closest-pair :: (point ∗ point) ⇒ point list ⇒ nat where
  t-find-closest-pair - [] = 1
| t-find-closest-pair - [-] = 1
| t-find-closest-pair (c₀, c₁) (p₀ # ps) = 1 + (
    let p₁ = find-closest p₀ (dist c₀ c₁) ps in
    t-find-closest p₀ (dist c₀ c₁) ps + (
    if dist c₀ c₁ ≤ dist p₀ p₁ then
```

```
    t-find-closest-pair (c₀, c₁) ps
  else
    t-find-closest-pair (p₀, p₁) ps
))

lemma t-find-closest-pair-eq-time-find-closest-pair-tm:
  t-find-closest-pair (c₀, c₁) ps = time (find-closest-pair-tm (c₀, c₁) ps)
  by (induction (c₀, c₁) ps arbitrary: c₀ c₁ rule: t-find-closest-pair.induct)
   (auto simp: time-simps find-closest-eq-val-find-closest-tm t-find-closest-eq-time-find-closest-tm)

lemma t-find-closest-pair-bound:
  assumes distinct ps sorted-snd ps δ = dist c₀ c₁ set ps = psL ∪ psR
  assumes ∀ p ∈ set ps. l − Δ < fst p ∧ fst p < l + Δ
  assumes ∀ p ∈ psL. fst p ≤ l ∀ p ∈ psR. l ≤ fst p
  assumes sparse Δ psL sparse Δ psR δ ≤ Δ
  shows t-find-closest-pair (c₀, c₁) ps ≤ 9 ∗ length ps + 1
  using assms
proof (induction (c₀, c₁) ps arbitrary: δ c₀ c₁ psL psR rule: t-find-closest-pair.induct)
  case (3 c₀ c₁ p₀ p₂ ps)
  let ?ps = p₂ # ps
  define p₁ where p₁-def: p₁ = find-closest p₀ (dist c₀ c₁) ?ps
  define PSL where PSL-def: PSL = psL − { p₀ }
  define PSR where PSR-def: PSR = psR − { p₀ }
  note defs = p₁-def PSL-def PSR-def
  have ∗: 0 ≤ Δ
    using 3.prems(3,10) zero-le-dist[of c₀ c₁] by argo
  hence t-find-closest p₀ Δ ?ps ≤ 8
    using t-find-closest-bound[of p₀ ?ps Δ psL psR] 3.prems by blast
  hence A: t-find-closest p₀ (dist c₀ c₁) ?ps ≤ 8
    by (metis 3.prems(3,10) order-trans t-find-closest-mono)
  have B: distinct ?ps sorted-snd ?ps
    using 3.prems(1,2) sorted-snd-def by simp-all
  have C: set ?ps = PSL ∪ PSR
    using defs 3.prems(1,4) by auto
  have D: ∀ p ∈ set ?ps. l − Δ < fst p ∧ fst p < l + Δ
    using 3.prems(5) by simp
  have E: ∀ p ∈ PSL. fst p ≤ l ∀ p ∈ PSR. l ≤ fst p
    using defs 3.prems(6,7) by simp-all
  have F: sparse Δ PSL sparse Δ PSR
    using defs 3.prems(8,9) sparse-def by simp-all
  show ?case
  proof (cases dist c₀ c₁ ≤ dist p₀ p₁)
    case True
    hence t-find-closest-pair (c₀, c₁) ?ps ≤ 9 ∗ length ?ps + 1
      using 3.hyps(1) 3.prems(3,10) defs(1) B C D E F by blast
    moreover have t-find-closest-pair (c₀, c₁) (p₀ # ?ps) =
                  1 + t-find-closest p₀ (dist c₀ c₁) ?ps + t-find-closest-pair (c₀, c₁)
?ps
      using defs True by (auto split: prod.splits)
```

```
    ultimately show ?thesis
      using A by auto
  next
    case False
    moreover have 0 ≤ dist p₀ p₁
      by auto
    ultimately have t-find-closest-pair (p₀, p₁) ?ps ≤ 9 * length ?ps + 1
      using 3.hyps(2) 3.prems(3,10) defs(1) B C D E F by auto
    moreover have t-find-closest-pair (c₀, c₁) (p₀ # ?ps) =
                  1 + t-find-closest p₀ (dist c₀ c₁) ?ps + t-find-closest-pair (p₀, p₁)
?ps
      using defs False by (auto split: prod.splits)
    ultimately show ?thesis
      using A by simp
  qed
qed auto

fun t-combine :: (point * point) ⇒ (point * point) ⇒ int ⇒ point list ⇒ nat where
  t-combine (p₀L, p₁L) (p₀R, p₁R) l ps = 1 + (
    let (c₀, c₁) = if dist p₀L p₁L < dist p₀R p₁R then (p₀L, p₁L) else (p₀R, p₁R) in
    let ps' = filter (λp. dist p (l, snd p) < dist c₀ c₁) ps in
    time (filter-tm (λp. dist p (l, snd p) < dist c₀ c₁) ps) + t-find-closest-pair (c₀,
c₁) ps'
  )

lemma t-combine-eq-time-combine-tm:
  t-combine (p₀L, p₁L) (p₀R, p₁R) l ps = time (combine-tm (p₀L, p₁L) (p₀R, p₁R)
l ps)
  by (auto simp: combine-tm.simps time-simps t-find-closest-pair-eq-time-find-closest-pair-tm
filter-eq-val-filter-tm)

lemma t-combine-bound:
  fixes ps :: point list
  assumes distinct ps sorted-snd ps set ps = psL ∪ psR
  assumes ∀ p ∈ psL. fst p ≤ l ∀ p ∈ psR. l ≤ fst p
  assumes sparse (dist p₀L p₁L) psL sparse (dist p₀R p₁R) psR
  shows t-combine (p₀L, p₁L) (p₀R, p₁R) l ps ≤ 10 * length ps + 3
proof −
  obtain c₀ c₁ where c-def:
    (c₀, c₁) = (if dist p₀L p₁L < dist p₀R p₁R then (p₀L, p₁L) else (p₀R, p₁R)) by
metis
  let ?P = (λp. dist p (l, snd p) < dist c₀ c₁)
  define ps' where ps'-def: ps' = filter ?P ps
  define psL' where psL'-def: psL' = { p ∈ psL. ?P p }
  define psR' where psR'-def: psR' = { p ∈ psR. ?P p }
  note defs = c-def ps'-def psL'-def psR'-def
  have sparse (dist c₀ c₁) psL sparse (dist c₀ c₁) psR
    using assms(6,7) sparse-mono c-def by (auto split: if-splits)
  hence sparse (dist c₀ c₁) psL' sparse (dist c₀ c₁) psR'
```

using ps_L'*-def* ps_R'*-def sparse-def* **by** *auto*
moreover have *distinct ps'*
using *ps'-def assms*(*1*) **by** *simp*
moreover have *sorted-snd ps'*
using *ps'-def assms*(*2*) *sorted-snd-def sorted-wrt-filter* **by** *blast*
moreover have $0 \leq dist\ c_0\ c_1$
by *simp*
moreover have $set\ ps' = ps_L' \cup ps_R'$
using *assms*(*3*) *defs*(*2*,*3*,*4*) *filter-Un* **by** *auto*
moreover have $\forall p \in set\ ps'.\ l - dist\ c_0\ c_1 < fst\ p \wedge fst\ p < l + dist\ c_0\ c_1$
using *ps'-def dist-transform* **by** *force*
moreover have $\forall p \in ps_L'.\ fst\ p \leq l\ \forall p \in ps_R'.\ l \leq fst\ p$
using *assms*(*4*,*5*) ps_L'*-def* ps_R'*-def* **by** *blast+*
ultimately have *t-find-closest-pair* $(c_0,\ c_1)\ ps' \leq 9 * length\ ps' + 1$
using *t-find-closest-pair-bound* **by** *blast*
moreover have $length\ ps' \leq length\ ps$
using *ps'-def* **by** *simp*
ultimately have $*$: *t-find-closest-pair* $(c_0,\ c_1)\ ps' \leq 9 * length\ ps + 1$
by *simp*
have *t-combine* $(p_{0L},\ p_{1L})\ (p_{0R},\ p_{1R})\ l\ ps =$
$1 + time\ (filter\text{-}tm\ ?P\ ps) + t\text{-}find\text{-}closest\text{-}pair\ (c_0,\ c_1)\ ps'$
using *defs* **by** (*auto split*: *prod.splits*)
also have $... = 2 + length\ ps + t\text{-}find\text{-}closest\text{-}pair\ (c_0,\ c_1)\ ps'$
using *time-filter-tm* **by** *auto*
finally show *?thesis*
using $*$ **by** *simp*
qed

declare *t-combine.simps* [*simp del*]

2.2.3 Divide and Conquer Algorithm

lemma *time-closest-pair-rec-tm-simps-1*:
assumes $length\ xs \leq 3$
shows $time\ (closest\text{-}pair\text{-}rec\text{-}tm\ xs) = 1 + time\ (length\text{-}tm\ xs) + time\ (mergesort\text{-}tm\ snd\ xs) + time\ (closest\text{-}pair\text{-}bf\text{-}tm\ xs)$
using *assms* **by** (*auto simp*: *time-simps length-eq-val-length-tm*)

lemma *time-closest-pair-rec-tm-simps-2*:
assumes $\neg\ (length\ xs \leq 3)$
shows $time\ (closest\text{-}pair\text{-}rec\text{-}tm\ xs) = 1 + (let\ (xs_L,\ xs_R) = val\ (split\text{-}at\text{-}tm\ (length\ xs\ div\ 2)\ xs)\ in\ let\ (ys_L,\ p_L) = val\ (closest\text{-}pair\text{-}rec\text{-}tm\ xs_L)\ in\ let\ (ys_R,\ p_R) = val\ (closest\text{-}pair\text{-}rec\text{-}tm\ xs_R)\ in\ let\ ys = val\ (merge\text{-}tm\ (\lambda p.\ snd\ p)\ ys_L\ ys_R)\ in\ time\ (length\text{-}tm\ xs) + time\ (split\text{-}at\text{-}tm\ (length\ xs\ div\ 2)\ xs) + time\ (closest\text{-}pair\text{-}rec\text{-}tm\ xs_L) + time\ (closest\text{-}pair\text{-}rec\text{-}tm\ xs_R) + time\ (merge\text{-}tm\ (\lambda p.\ snd\ p)\ ys_L\ ys_R) + t\text{-}combine\ p_L\ p_R\ (fst\ (hd\ xs_R))\ ys$

```
  )
  using assms
  apply (subst closest-pair-rec-tm.simps)
  by (auto simp del: closest-pair-rec-tm.simps
        simp add: time-simps length-eq-val-length-tm t-combine-eq-time-combine-tm
            split: prod.split)

function closest-pair-recurrence :: nat ⇒ real where
  n ≤ 3 ⟹ closest-pair-recurrence n = 3 + n + mergesort-recurrence n + n * n
| 3 < n ⟹ closest-pair-recurrence n = 7 + 13 * n +
    closest-pair-recurrence (nat ⌊real n / 2⌋) + closest-pair-recurrence (nat ⌈real n
/ 2⌉)
  by force simp-all
termination by akra-bazzi-termination simp-all

lemma closest-pair-recurrence-nonneg[simp]:
  0 ≤ closest-pair-recurrence n
  by (induction n rule: closest-pair-recurrence.induct) auto

lemma time-closest-pair-rec-conv-closest-pair-recurrence:
  assumes distinct ps sorted-fst ps
  shows time (closest-pair-rec-tm ps) ≤ closest-pair-recurrence (length ps)
  using assms
proof (induction ps rule: length-induct)
  case (1 ps)
  let ?n = length ps
  show ?case
  proof (cases ?n ≤ 3)
    case True
   hence time (closest-pair-rec-tm ps) = 1 + time (length-tm ps) + time (mergesort-tm
snd ps) + time (closest-pair-bf-tm ps)
      using time-closest-pair-rec-tm-simps-1 by simp
    moreover have closest-pair-recurrence ?n = 3 + ?n + mergesort-recurrence
?n + ?n * ?n
      using True by simp
    moreover have time (length-tm ps) ≤ 1 + ?n time (mergesort-tm snd ps) ≤
mergesort-recurrence ?n
                time (closest-pair-bf-tm ps) ≤ 1 + ?n * ?n
      using time-length-tm[of ps] time-mergesort-conv-mergesort-recurrence[of snd
ps] time-closest-pair-bf-tm[of ps] by auto
    ultimately show ?thesis
      by linarith
  next
    case False

    obtain XS_L XS_R where XS-def: (XS_L, XS_R) = val (split-at-tm (?n div 2)
ps)
      using prod.collapse by blast
   obtain YS_L C_0L C_1L where CP_L-def: (YS_L, C_0L, C_1L) = val (closest-pair-rec-tm
```

XS_L)
using *prod.collapse* **by** *metis*
obtain YS_R C_{0R} C_{1R} **where** CP_R*-def*: (YS_R, C_{0R}, C_{1R}) = *val* (*closest-pair-rec-tm* XS_R)
using *prod.collapse* **by** *metis*
define *YS* **where** *YS* = *val* (*merge-tm* (λ*p*. *snd p*) YS_L YS_R)
obtain C_0 C_1 **where** C_{01}*-def*: (C_0, C_1) = *val* (*combine-tm* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (*fst* (*hd* XS_R)) *YS*)
using *prod.collapse* **by** *metis*
note *defs* = *XS-def* CP_L*-def* CP_R*-def YS-def* C_{01}*-def*

have *XSLR*: XS_L = *take* (*?n div 2*) *ps* XS_R = *drop* (*?n div 2*) *ps*
using *defs* **by** (*auto simp*: *split-at-take-drop-conv split-at-eq-val-split-at-tm*)
hence *length* XS_L = *?n div 2 length* XS_R = *?n* − *?n div 2*
by *simp-all*
hence ∗: (*nat* ⌊*real ?n* / *2*⌋) = *length* XS_L (*nat* ⌈*real ?n* / *2*⌉) = *length* XS_R
by *linarith*+
have *length* XS_L = *length* YS_L *length* XS_R = *length* YS_R
using *defs closest-pair-rec-set-length-sorted-snd closest-pair-rec-eq-val-closest-pair-rec-tm* **by** *metis*+
hence *L*: *?n* = *length* YS_L + *length* YS_R
using *defs XSLR* **by** *fastforce*

have *1* < *length* XS_L *length* XS_L < *length ps*
using *False XSLR* **by** *simp-all*
moreover have *distinct* XS_L *sorted-fst* XS_L
using *XSLR 1.prems*(*1*,*2*) *sorted-fst-def sorted-wrt-take* **by** *simp-all*
ultimately have *time* (*closest-pair-rec-tm* XS_L) ≤ *closest-pair-recurrence* (*length* XS_L)
using *1.IH* **by** *simp*
hence *IHL*: *time* (*closest-pair-rec-tm* XS_L) ≤ *closest-pair-recurrence* (*nat* ⌊*real ?n* / *2*⌋)
using ∗ **by** *simp*

have *1* < *length* XS_R *length* XS_R < *length ps*
using *False XSLR* **by** *simp-all*
moreover have *distinct* XS_R *sorted-fst* XS_R
using *XSLR 1.prems*(*1*,*2*) *sorted-fst-def sorted-wrt-drop* **by** *simp-all*
ultimately have *time* (*closest-pair-rec-tm* XS_R) ≤ *closest-pair-recurrence* (*length* XS_R)
using *1.IH* **by** *simp*
hence *IHR*: *time* (*closest-pair-rec-tm* XS_R) ≤ *closest-pair-recurrence* (*nat* ⌈*real ?n* / *2*⌉)
using ∗ **by** *simp*

have (*YS*, C_0, C_1) = *val* (*closest-pair-rec-tm ps*)
using *False closest-pair-rec-simps defs* **by** (*auto simp*: *Let-def length-eq-val-length-tm split*!: *prod.split*)
hence *set ps* = *set YS length ps* = *length YS distinct YS sorted-snd YS*

using *1.prems closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct*
closest-pair-rec-eq-val-closest-pair-rec-tm **by** *auto*
moreover have $\forall\, p \in set\ YS_L.\ fst\ p \leq fst\ (hd\ XS_R)$
using *False 1.prems(2) XSLR* ‹*length* $XS_L <$ *length ps*› ‹*length* $XS_L =$ *length ps div 2*›
CP_L*-def sorted-fst-take-less-hd-drop closest-pair-rec-set-length-sorted-snd*
closest-pair-rec-eq-val-closest-pair-rec-tm **by** *metis*
moreover have $\forall\, p \in set\ YS_R.\ fst\ (hd\ XS_R) \leq fst\ p$
using *False 1.prems(2) XSLR* CP_R*-def sorted-fst-hd-drop-less-drop*
closest-pair-rec-set-length-sorted-snd closest-pair-rec-eq-val-closest-pair-rec-tm
by *metis*
moreover have *set YS* = *set* $YS_L \cup$ *set* YS_R
using *set-merge defs* **by** (*metis merge-eq-val-merge-tm*)
moreover have *sparse* (*dist* C_{0L} C_{1L}) (*set* YS_L)
using CP_L*-def* ‹*1 <* length XS_L› ‹*distinct* XS_L› ‹*sorted-fst* XS_L›
closest-pair-rec-dist closest-pair-rec-set-length-sorted-snd
closest-pair-rec-eq-val-closest-pair-rec-tm **by** *auto*
moreover have *sparse* (*dist* C_{0R} C_{1R}) (*set* YS_R)
using CP_R*-def* ‹*1 <* length XS_R› ‹*distinct* XS_R› ‹*sorted-fst* XS_R›
closest-pair-rec-dist closest-pair-rec-set-length-sorted-snd
closest-pair-rec-eq-val-closest-pair-rec-tm **by** *auto*
ultimately have *combine-bound*: *t-combine* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (*fst* (*hd* XS_R)) *YS* $\leq$ *3 + 10 * ?n*
using *t-combine-bound*[*of YS set* YS_L *set* YS_R *fst* (*hd* XS_R)] **by** (*simp add*: *add.commute*)
have *time* (*closest-pair-rec-tm ps*) = *1 + time* (*length-tm ps*) + *time* (*split-at-tm* (*?n div 2*) *ps*) +
time (*closest-pair-rec-tm* XS_L) + *time* (*closest-pair-rec-tm* XS_R) + *time* (*merge-tm* (*λp. snd p*) YS_L YS_R) +
t-combine (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (*fst* (*hd* XS_R)) *YS*
using *time-closest-pair-rec-tm-simps-2*[*OF False*] *defs*
by (*auto simp del*: *closest-pair-rec-tm.simps simp add*: *Let-def split*: *prod.split*)
also have ... $\leq$ *7 + 13 * ?n + time* (*closest-pair-rec-tm* XS_L) + *time* (*closest-pair-rec-tm* XS_R)
using *time-merge-tm*[*of* (*λp. snd p*) YS_L YS_R] *L combine-bound* **by** (*simp add*: *time-length-tm time-split-at-tm*)
also have ... $\leq$ *7 + 13 * ?n + closest-pair-recurrence* (*nat* $\lfloor$*real ?n / 2*$\rfloor$) + *closest-pair-recurrence* (*nat* $\lceil$*real ?n / 2*$\rceil$)
using *IHL IHR* **by** *simp*
also have ... = *closest-pair-recurrence* (*length ps*)
using *False* **by** *simp*
finally show *?thesis*
by *simp*
qed
qed

theorem *closest-pair-recurrence*:
closest-pair-recurrence $\in \Theta(\lambda n.\ n * ln\ n)$
by (*master-theorem*) *auto*

theorem *time-closest-pair-rec-bigo*:
(λ*xs*. *time* (*closest-pair-rec-tm xs*)) ∈ *O*[*length going-to at-top within* { *ps*. *distinct ps* ∧ *sorted-fst ps* }]((λ*n*. *n* ∗ *ln n*) *o length*)
proof −
have *0*: ⋀*ps*. *ps* ∈ { *ps*. *distinct ps* ∧ *sorted-fst ps* } ⟹
time (*closest-pair-rec-tm ps*) ≤ (*closest-pair-recurrence o length*) *ps*
unfolding *comp-def* **using** *time-closest-pair-rec-conv-closest-pair-recurrence* **by** *auto*
show *?thesis*
using *bigo-measure-trans*[*OF 0*] *bigthetaD1*[*OF closest-pair-recurrence*] *of-nat-0-le-iff* **by** *blast*
qed

definition *closest-pair-time* :: *nat* ⇒ *real* **where**
closest-pair-time n = *1* + *mergesort-recurrence n* + *closest-pair-recurrence n*

lemma *time-closest-pair-conv-closest-pair-recurrence*:
assumes *distinct ps*
shows *time* (*closest-pair-tm ps*) ≤ *closest-pair-time* (*length ps*)
using *assms*
unfolding *closest-pair-time-def*
proof (*induction rule*: *induct-list012*)
case (*3 x y zs*)
let *?ps* = *x* # *y* # *zs*
define *xs* **where** *xs* = *val* (*mergesort-tm fst ?ps*)
have ∗: *distinct xs sorted-fst xs length xs* = *length ?ps*
using *xs-def mergesort*(*4*)[*OF 3.prems*, *of fst*] *mergesort*(*1*)[*of fst ?ps*] *mergesort*(*3*)[*of fst ?ps*]
sorted-fst-def mergesort-eq-val-mergesort-tm **by** *metis*+
have *time* (*closest-pair-tm ?ps*) = *1* + *time* (*mergesort-tm fst ?ps*) + *time* (*closest-pair-rec-tm xs*)
using *xs-def* **by** (*auto simp del*: *mergesort-tm.simps closest-pair-rec-tm.simps simp add*: *time-simps split*: *prod.split*)
also have ... ≤ *1* + *mergesort-recurrence* (*length ?ps*) + *time* (*closest-pair-rec-tm xs*)
using *time-mergesort-conv-mergesort-recurrence*[*of fst ?ps*] **by** *simp*
also have ... ≤ *1* + *mergesort-recurrence* (*length ?ps*) + *closest-pair-recurrence* (*length ?ps*)
using *time-closest-pair-rec-conv-closest-pair-recurrence*[*of xs*] ∗ **by** *auto*
finally show *?case*
by *blast*
qed (*auto simp*: *time-simps*)

corollary *closest-pair-time*:
closest-pair-time ∈ *O*(λ*n*. *n* ∗ *ln n*)
unfolding *closest-pair-time-def*
using *mergesort-recurrence closest-pair-recurrence sum-in-bigo*(*1*) *const-1-bigo-n-ln-n* **by** *blast*

corollary *time-closest-pair-bigo*:
(λ*ps*. *time* (*closest-pair-tm ps*)) ∈ *O*[*length going-to at-top within* { *ps*. *distinct ps* }]((λ*n*. *n* ∗ *ln n*) *o length*)
proof −
have *0*: ⋀*ps*. *ps* ∈ { *ps*. *distinct ps* } ⟹
time (*closest-pair-tm ps*) ≤ (*closest-pair-time o length*) *ps*
unfolding *comp-def* **using** *time-closest-pair-conv-closest-pair-recurrence* **by** *auto*
show *?thesis*
using *bigo-measure-trans*[*OF 0*] *closest-pair-time* **by** *fastforce*
qed

2.3 Code Export

2.3.1 Combine Step

fun *find-closest-code* :: *point* ⇒ *int* ⇒ *point list* ⇒ (*int* ∗ *point*) **where**
find-closest-code - - [] = *undefined*
| *find-closest-code p* - [p_0] = (*dist-code p* p_0, p_0)
| *find-closest-code p* δ (p_0 # *ps*) = (
let δ_0 = *dist-code p* p_0 *in*
if δ ≤ (*snd* p_0 − *snd p*)^2 *then*
(δ_0, p_0)
else
let (δ_1, p_1) = *find-closest-code p* (*min* δ δ_0) *ps in*
if δ_0 ≤ δ_1 *then*
(δ_0, p_0)
else
(δ_1, p_1)
)

lemma *find-closest-code-dist-eq*:
0 < *length ps* ⟹ (δ_c, *c*) = *find-closest-code p* δ *ps* ⟹ δ_c = *dist-code p c*
proof (*induction p* δ *ps arbitrary*: δ_c *c rule*: *find-closest-code.induct*)
case (*3 p* δ p_0 p_2 *ps*)
show *?case*
proof *cases*
assume δ ≤ (*snd* p_0 − *snd p*)^2
thus *?thesis*
using *3.prems*(*2*) **by** *simp*
next
assume *A*: ¬ δ ≤ (*snd* p_0 − *snd p*)^2
define δ_0 **where** δ_0-*def*: δ_0 = *dist-code p* p_0
obtain δ_1 p_1 **where** δ_1-*def*: (δ_1, p_1) = *find-closest-code p* (*min* δ δ_0) (p_2 # *ps*)
by (*metis surj-pair*)
note *defs* = δ_0-*def* δ_1-*def*
have δ_1 = *dist-code p* p_1
using *3.IH*[*of* δ_0 δ_1 p_1] *A defs* **by** *simp*
thus *?thesis*

```
    using defs 3.prems by (auto simp: Let-def split: if-splits prod.splits)
  qed
qed simp-all

declare find-closest.simps [simp add]

lemma find-closest-code-eq:
  assumes 0 < length ps δ = dist c₀ c₁ δ' = dist-code c₀ c₁ sorted-snd (p # ps)
  assumes c = find-closest p δ ps (δ_c', c') = find-closest-code p δ' ps
  shows c = c'
  using assms
proof (induction p δ ps arbitrary: δ' c₀ c₁ c δ_c' c' rule: find-closest.induct)
  case (3 p δ p₀ p₂ ps)
  define δ₀ δ₀' where δ₀-def: δ₀ = dist p p₀ δ₀' = dist-code p p₀
  obtain p₁ δ₁' p₁' where δ₁-def: p₁ = find-closest p (min δ δ₀) (p₂ # ps)
    (δ₁', p₁') = find-closest-code p (min δ' δ₀') (p₂ # ps)
    by (metis surj-pair)
  note defs = δ₀-def δ₁-def
  show ?case
  proof cases
    assume *: δ ≤ snd p₀ − snd p
    hence δ' ≤ (snd p₀ − snd p)²
      using 3.prems(2,3) dist-eq-dist-code-abs-le by fastforce
    thus ?thesis
      using * 3.prems(5,6) by simp
  next
    assume *: ¬ δ ≤ snd p₀ − snd p
    moreover have 0 ≤ snd p₀ − snd p
      using 3.prems(4) sorted-snd-def by simp
    ultimately have A: ¬ δ' ≤ (snd p₀ − snd p)²
      using 3.prems(2,3) dist-eq-dist-code-abs-le[of c₀ c₁ snd p₀ − snd p] by simp
    have min δ δ₀ = δ ⟷ min δ' δ₀' = δ' min δ δ₀ = δ₀ ⟷ min δ' δ₀' = δ₀'
     by (metis 3.prems(2,3) defs(1,2) dist-eq-dist-code-le min.commute min-def)+
    moreover have sorted-snd (p # p₂ # ps)
      using 3.prems(4) sorted-snd-def by simp
    ultimately have B: p₁ = p₁'
      using 3.IH[of c₀ c₁ δ' p₁ δ₁' p₁'] 3.IH[of p p₀ δ₀' p₁ δ₁' p₁'] 3.prems(2,3)
defs * by auto
    have δ₁' = dist-code p p₁'
      using find-closest-code-dist-eq defs by blast
    hence δ₀ ≤ dist p p₁ ⟷ δ₀' ≤ δ₁'
      using defs(1,2) dist-eq-dist-code-le by (simp add: B)
    thus ?thesis
      using 3.prems(5,6) * A B defs by (auto simp: Let-def split: prod.splits)
  qed
qed auto

fun find-closest-pair-code :: (int * point * point) ⇒ point list ⇒ (int * point *
point) where
```

```
  find-closest-pair-code (δ, c0, c1) [] = (δ, c0, c1)
| find-closest-pair-code (δ, c0, c1) [p] = (δ, c0, c1)
| find-closest-pair-code (δ, c0, c1) (p0 # ps) = (
    let (δ', p1) = find-closest-code p0 δ ps in
    if δ ≤ δ' then
      find-closest-pair-code (δ, c0, c1) ps
    else
      find-closest-pair-code (δ', p0, p1) ps
  )

lemma find-closest-pair-code-dist-eq:
  assumes δ = dist-code c0 c1 (Δ, C0, C1) = find-closest-pair-code (δ, c0, c1) ps
  shows Δ = dist-code C0 C1
  using assms
proof (induction (δ, c0, c1) ps arbitrary: δ c0 c1 Δ C0 C1 rule: find-closest-pair-code.induct)
  case (3 δ c0 c1 p0 p2 ps)
  obtain δ' p1 where δ'-def: (δ', p1) = find-closest-code p0 δ (p2 # ps)
    by (metis surj-pair)
  hence A: δ' = dist-code p0 p1
    using find-closest-code-dist-eq by blast
  show ?case
  proof (cases δ ≤ δ')
    case True
    obtain Δ' C0' C1' where Δ'-def: (Δ', C0', C1') = find-closest-pair-code (δ,
c0, c1) (p2 # ps)
      by (metis prod-cases4)
    note defs = δ'-def Δ'-def
    hence Δ' = dist-code C0' C1'
      using 3.hyps(1)[of (δ', p1) δ' p1] 3.prems(1) True δ'-def by blast
    moreover have Δ = Δ' C0 = C0' C1 = C1'
    using defs True 3.prems(2) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by simp
  next
    case False
    obtain Δ' C0' C1' where Δ'-def: (Δ', C0', C1') = find-closest-pair-code (δ',
p0, p1) (p2 # ps)
      by (metis prod-cases4)
    note defs = δ'-def Δ'-def
    hence Δ' = dist-code C0' C1'
      using 3.hyps(2)[of (δ', p1) δ' p1] A False δ'-def by blast
    moreover have Δ = Δ' C0 = C0' C1 = C1'
    using defs False 3.prems(2) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by simp
  qed
qed auto

declare find-closest-pair.simps [simp add]
```

lemma *find-closest-pair-code-eq*:
assumes δ = *dist* c_0 c_1 δ' = *dist-code* c_0 c_1 *sorted-snd ps*
assumes (C_0, C_1) = *find-closest-pair* (c_0, c_1) *ps*
assumes (Δ', C_0', C_1') = *find-closest-pair-code* (δ', c_0, c_1) *ps*
shows $C_0 = C_0' \wedge C_1 = C_1'$
using *assms*
proof (*induction* (c_0, c_1) *ps arbitrary*: δ δ' c_0 c_1 C_0 C_1 Δ' C_0' C_1' *rule*: *find-closest-pair.induct*)
case (*3* c_0 c_1 p_0 p_2 *ps*)
obtain p_1 δ_p' p_1' **where** δ_p-*def*: p_1 = *find-closest* p_0 δ (p_2 # *ps*)
(δ_p', p_1') = *find-closest-code* p_0 δ' (p_2 # *ps*)
by (*metis surj-pair*)
hence *A*: δ_p' = *dist-code* p_0 p_1'
using *find-closest-code-dist-eq* **by** *blast*
have *B*: $p_1 = p_1'$
using *3.prems*(*1*,*2*,*3*) δ_p-*def find-closest-code-eq* **by** *blast*
show *?case*
proof (*cases* $\delta \leq$ *dist* p_0 p_1)
case *True*
hence *C*: $\delta' \leq \delta_p'$
by (*simp add*: *3.prems*(*1*,*2*) *A B dist-eq-dist-code-le*)
obtain C_{0i} C_{1i} Δ_i' C_{0i}' C_{1i}' **where** Δ_i-*def*:
(C_{0i}, C_{1i}) = *find-closest-pair* (c_0, c_1) (p_2 # *ps*)
(Δ_i', C_{0i}', C_{1i}') = *find-closest-pair-code* (δ', c_0, c_1) (p_2 # *ps*)
by (*metis prod-cases3*)
note *defs* = δ_p-*def* Δ_i-*def*
have *sorted-snd* (p_2 # *ps*)
using *3.prems*(*3*) *sorted-snd-def* **by** *simp*
hence $C_{0i} = C_{0i}' \wedge C_{1i} = C_{1i}'$
using *3.hyps*(*1*) *3.prems*(*1*,*2*) *True defs* **by** *blast*
moreover have $C_0 = C_{0i}$ $C_1 = C_{1i}$
using *defs*(*1*,*3*) *True 3.prems*(*1*,*4*) **apply** (*auto split*: *prod.splits*) **by** (*metis Pair-inject*)+
moreover have $\Delta' = \Delta_i'$ $C_0' = C_{0i}'$ $C_1' = C_{1i}'$
using *defs*(*2*,*4*) *C 3.prems*(*5*) **apply** (*auto split*: *prod.splits*) **by** (*metis Pair-inject*)+
ultimately show *?thesis*
by *simp*
next
case *False*
hence *C*: $\neg\ \delta' \leq \delta_p'$
by (*simp add*: *3.prems*(*1*,*2*) *A B dist-eq-dist-code-le*)
obtain C_{0i} C_{1i} Δ_i' C_{0i}' C_{1i}' **where** Δ_i-*def*:
(C_{0i}, C_{1i}) = *find-closest-pair* (p_0, p_1) (p_2 # *ps*)
(Δ_i', C_{0i}', C_{1i}') = *find-closest-pair-code* (δ_p', p_0, p_1') (p_2 # *ps*)
by (*metis prod-cases3*)
note *defs* = δ_p-*def* Δ_i-*def*
have *sorted-snd* (p_2 # *ps*)
using *3.prems*(*3*) *sorted-snd-def* **by** *simp*

hence $C_{0i} = C_{0i}' \wedge C_{1i} = C_{1i}'$
using *3.prems(1) 3.hyps(2) A B False defs* **by** *blast*
moreover have $C_0 = C_{0i}\ C_1 = C_{1i}$
using *defs(1,3) False 3.prems(1,4)* **apply** (*auto split*: *prod.splits*) **by** (*metis Pair-inject*)+
moreover have $\Delta' = \Delta_i'\ C_0' = C_{0i}'\ C_1' = C_{1i}'$
using *defs(2,4) C 3.prems(5)* **apply** (*auto split*: *prod.splits*) **by** (*metis Pair-inject*)+
ultimately show *?thesis*
by *simp*
qed
qed *auto*

fun *combine-code* :: (*int* $*$ *point* $*$ *point*) $\Rightarrow$ (*int* $*$ *point* $*$ *point*) $\Rightarrow$ *int* $\Rightarrow$ *point list* $\Rightarrow$ (*int* $*$ *point* $*$ *point*) **where**
combine-code (δ_L, p_{0L}, p_{1L}) (δ_R, p_{0R}, p_{1R}) *l ps* = (
let (δ, c_0, c_1) = *if* $\delta_L < \delta_R$ *then* (δ_L, p_{0L}, p_{1L}) *else* (δ_R, p_{0R}, p_{1R}) *in*
let ps' = *filter* (λp. (*fst p* $-$ *l*)$^2 < \delta$) *ps in*
find-closest-pair-code (δ, c_0, c_1) *ps'*
)

lemma *combine-code-dist-eq*:
assumes δ_L = *dist-code* p_{0L} p_{1L} δ_R = *dist-code* p_{0R} p_{1R}
assumes (δ, c_0, c_1) = *combine-code* (δ_L, p_{0L}, p_{1L}) (δ_R, p_{0R}, p_{1R}) *l ps*
shows δ = *dist-code* c_0 c_1
using *assms* **by** (*auto simp*: *find-closest-pair-code-dist-eq split*: *if-splits*)

lemma *combine-code-eq*:
assumes δ_L' = *dist-code* p_{0L} p_{1L} δ_R' = *dist-code* p_{0R} p_{1R} *sorted-snd ps*
assumes (c_0, c_1) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
assumes (δ', c_0', c_1') = *combine-code* (δ_L', p_{0L}, p_{1L}) (δ_R', p_{0R}, p_{1R}) *l ps*
shows $c_0 = c_0' \wedge c_1 = c_1'$
proof $-$
obtain C_{0i} C_{1i} Δ_i' C_{0i}' C_{1i}' **where** Δ_i-*def*:
(C_{0i}, C_{1i}) = (*if dist* p_{0L} p_{1L} $<$ *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}))
(Δ_i', C_{0i}', C_{1i}') = (*if* $\delta_L' < \delta_R'$ *then* (δ_L', p_{0L}, p_{1L}) *else* (δ_R', p_{0R}, p_{1R}))
by *metis*
define *ps' ps''* **where** *ps'-def*:
ps' = *filter* (λp. *dist p* (*l*, *snd p*) $<$ *dist* C_{0i} C_{1i}) *ps*
ps'' = *filter* (λp. (*fst p* $-$ *l*)$^2 < \Delta_i'$) *ps*
obtain C_0 C_1 Δ' C_0' C_1' **where** Δ-*def*:
(C_0, C_1) = *find-closest-pair* (C_{0i}, C_{1i}) *ps'*
(Δ', C_0', C_1') = *find-closest-pair-code* (Δ_i', C_{0i}', C_{1i}') *ps''*
by (*metis prod-cases3*)
note *defs* = Δ_i-*def ps'-def* Δ-*def*
have $*$: $C_{0i} = C_{0i}'\ C_{1i} = C_{1i}'\ \Delta_i' =$ *dist-code* C_{0i}' C_{1i}'
using Δ_i-*def assms(1,2,3,4) dist-eq-dist-code-lt* **by** (*auto split*: *if-splits*)
hence $\bigwedge p$. |*fst p* $-$ *l*| $<$ *dist* C_{0i} C_{1i} $\longleftrightarrow$ (*fst p* $-$ *l*)$^2 < \Delta_i'$
using *dist-eq-dist-code-abs-lt* **by** (*metis* (*mono-tags*) *of-int-abs*)

hence $ps' = ps''$
using *ps'-def dist-fst-abs* **by** *auto*
moreover have *sorted-snd ps'*
using *assms(3) ps'-def sorted-snd-def sorted-wrt-filter* **by** *blast*
ultimately have $C_0 = C_0'\ C_1 = C_1'$
using $*$ *find-closest-pair-code-eq* Δ*-def* **by** *blast+*
moreover have $C_0 = c_0\ C_1 = c_1$
using *assms(4) defs(1,3,5)* **apply** (*auto simp*: *combine.simps split*: *prod.splits*)
by (*metis Pair-inject*)+
moreover have $C_0' = c_0'\ C_1' = c_1'$
using *assms(5) defs(2,4,6)* **apply** (*auto split*: *prod.splits*) **by** (*metis prod.inject*)+
ultimately show *?thesis*
by *blast*
qed

2.3.2 Divide and Conquer Algorithm

function *closest-pair-rec-code* :: *point list* $\Rightarrow$ (*point list* $*$ *int* $*$ *point* $*$ *point*)
where
closest-pair-rec-code xs = (
let n = *length xs in*
if n $\leq$ *3 then*
(*mergesort snd xs*, *closest-pair-bf-code xs*)
else
let (xs_L, xs_R) = *split-at* (*n div 2*) *xs in*
let l = *fst* (*hd* xs_R) *in*

let (ys_L, p_L) = *closest-pair-rec-code* xs_L *in*
let (ys_R, p_R) = *closest-pair-rec-code* xs_R *in*

let ys = *merge snd* ys_L ys_R *in*
(*ys*, *combine-code* p_L p_R *l ys*)
)
by *pat-completeness auto*
termination *closest-pair-rec-code*
by (*relation Wellfounded.measure* (λ*xs. length xs*))
(*auto simp*: *split-at-take-drop-conv Let-def*)

lemma *closest-pair-rec-code-simps*:
assumes *n* = *length xs* $\neg$ (*n* $\leq$ *3*)
shows *closest-pair-rec-code xs* = (
let (xs_L, xs_R) = *split-at* (*n div 2*) *xs in*
let l = *fst* (*hd* xs_R) *in*
let (ys_L, p_L) = *closest-pair-rec-code* xs_L *in*
let (ys_R, p_R) = *closest-pair-rec-code* xs_R *in*
let ys = *merge snd* ys_L ys_R *in*
(*ys*, *combine-code* p_L p_R *l ys*)
)
using *assms* **by** (*auto simp*: *Let-def*)

declare *combine.simps combine-code.simps closest-pair-rec-code.simps* [*simp del*]

lemma *closest-pair-rec-code-dist-eq*:
assumes *1 < length xs* (*ys*, δ, c_0, c_1) = *closest-pair-rec-code xs*
shows δ = *dist-code* c_0 c_1
using *assms*
proof (*induction xs arbitrary*: *ys* δ c_0 c_1 *rule*: *length-induct*)
case (*1 xs*)
let *?n* = *length xs*
show *?case*
proof (*cases ?n* $\leq$ *3*)
case *True*
hence (δ, c_0, c_1) = *closest-pair-bf-code xs*
using *1.prems*(*2*) *closest-pair-rec-code.simps* **by** *simp*
thus *?thesis*
using *1.prems*(*1*) *closest-pair-bf-code-dist-eq* **by** *simp*
next
case *False*

obtain XS_L XS_R **where** XS_{LR}-*def*: (XS_L, XS_R) = *split-at* (*?n div 2*) *xs*
using *prod.collapse* **by** *blast*
define *L* **where** *L* = *fst* (*hd* XS_R)

obtain YS_L Δ_L C_{0L} C_{1L} **where** YSC_{01L}-*def*: (YS_L, Δ_L, C_{0L}, C_{1L}) = *closest-pair-rec-code* XS_L
using *prod.collapse* **by** *metis*
obtain YS_R Δ_R C_{0R} C_{1R} **where** YSC_{01R}-*def*: (YS_R, Δ_R, C_{0R}, C_{1R}) = *closest-pair-rec-code* XS_R
using *prod.collapse* **by** *metis*

define *YS* **where** *YS* = *merge* (λ*p*. *snd p*) YS_L YS_R
obtain Δ C_0 C_1 **where** C_{01}-*def*: (Δ, C_0, C_1) = *combine-code* (Δ_L, C_{0L}, C_{1L}) (Δ_R, C_{0R}, C_{1R}) *L YS*
using *prod.collapse* **by** *metis*
note *defs* = XS_{LR}-*def L-def* YSC_{01L}-*def* YSC_{01R}-*def YS-def* C_{01}-*def*

have *1 < length* XS_L *length* XS_L *< length xs*
using *False 1.prems*(*1*) *defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHL*: Δ_L = *dist-code* C_{0L} C_{1L}
using *1.IH defs* **by** *metis+*

have *1 < length* XS_R *length* XS_R *< length xs*
using *False 1.prems*(*1*) *defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHR*: Δ_R = *dist-code* C_{0R} C_{1R}
using *1.IH defs* **by** *metis+*

have $*$: (*YS*, Δ, C_0, C_1) = *closest-pair-rec-code xs*
using *False closest-pair-rec-code-simps defs* **by** (*auto simp*: *Let-def split*:

prod.split)
    moreover have Δ = *dist-code* C_0 C_1
      using *combine-code-dist-eq IHL IHR* C_{01}*-def* **by** *blast*
    ultimately show *?thesis*
      using *1.prems*(*2*) ∗ **by** (*metis Pair-inject*)
  qed
qed

lemma *closest-pair-rec-ys-eq*:
  assumes *1* < *length xs*
  assumes (*ys*, c_0, c_1) = *closest-pair-rec xs*
  assumes (*ys'*, δ', c_0', c_1') = *closest-pair-rec-code xs*
  shows *ys* = *ys'*
  using *assms*
proof (*induction xs arbitrary*: *ys* c_0 c_1 *ys'* δ' c_0' c_1' *rule*: *length-induct*)
  case (*1 xs*)
  let *?n* = *length xs*
  show *?case*
  proof (*cases ?n* ≤ *3*)
    case *True*
    hence *ys* = *mergesort snd xs*
      using *1.prems*(*2*) *closest-pair-rec.simps* **by** *simp*
    moreover have *ys'* = *mergesort snd xs*
      using *1.prems*(*3*) *closest-pair-rec-code.simps* **by** (*simp add*: *True*)
    ultimately show *?thesis*
      using *1.prems*(*1*) **by** *simp*
  next
    case *False*

    obtain XS_L XS_R **where** XS_{LR}*-def*: (XS_L, XS_R) = *split-at* (*?n div 2*) *xs*
      using *prod.collapse* **by** *blast*
    define *L* **where** *L* = *fst* (*hd* XS_R)

    obtain YS_L C_{0L} C_{1L} YS_L' Δ_L' C_{0L}' C_{1L}' **where** YSC_{01L}*-def*:
      (YS_L, C_{0L}, C_{1L}) = *closest-pair-rec* XS_L
      (YS_L', Δ_L', C_{0L}', C_{1L}') = *closest-pair-rec-code* XS_L
      using *prod.collapse* **by** *metis*
    obtain YS_R C_{0R} C_{1R} YS_R' Δ_R' C_{0R}' C_{1R}' **where** YSC_{01R}*-def*:
      (YS_R, C_{0R}, C_{1R}) = *closest-pair-rec* XS_R
      (YS_R', Δ_R', C_{0R}', C_{1R}') = *closest-pair-rec-code* XS_R
      using *prod.collapse* **by** *metis*

    define *YS YS'* **where** *YS-def*:
      YS = *merge* (λ*p*. *snd p*) YS_L YS_R
      YS' = *merge* (λ*p*. *snd p*) YS_L' YS_R'
    obtain C_0 C_1 Δ' C_0' C_1' **where** C_{01}*-def*:
      (C_0, C_1) = *combine* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) *L YS*
      (Δ', C_0', C_1') = *combine-code* (Δ_L', C_{0L}', C_{1L}') (Δ_R', C_{0R}', C_{1R}') *L YS'*
      using *prod.collapse* **by** *metis*

```
    note defs = XS_LR-def L-def YSC_01L-def YSC_01R-def YS-def C_01-def

    have 1 < length XS_L length XS_L < length xs
      using False 1.prems(1) defs by (auto simp: split-at-take-drop-conv)
    hence IHL: YS_L = YS_L'
      using 1.IH defs by metis

    have 1 < length XS_R length XS_R < length xs
      using False 1.prems(1) defs by (auto simp: split-at-take-drop-conv)
    hence IHR: YS_R = YS_R'
      using 1.IH defs by metis

    have (YS, C_0, C_1) = closest-pair-rec xs
      using False closest-pair-rec-simps defs(1,2,3,5,7,9)
      by (auto simp: Let-def split: prod.split)
    moreover have (YS', Δ', C_0', C_1') = closest-pair-rec-code xs
      using False closest-pair-rec-code-simps defs(1,2,4,6,8,10)
      by (auto simp: Let-def split: prod.split)
    moreover have YS = YS'
      using IHL IHR YS-def by simp
    ultimately show ?thesis
      by (metis 1.prems(2,3) Pair-inject)
  qed
qed

lemma closest-pair-rec-code-eq:
  assumes 1 < length xs
  assumes (ys, c_0, c_1) = closest-pair-rec xs
  assumes (ys', δ', c_0', c_1') = closest-pair-rec-code xs
  shows c_0 = c_0' ∧ c_1 = c_1'
  using assms
proof (induction xs arbitrary: ys c_0 c_1 ys' δ' c_0' c_1' rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence (c_0, c_1) = closest-pair-bf xs
      using 1.prems(2) closest-pair-rec.simps by simp
    moreover have (δ', c_0', c_1') = closest-pair-bf-code xs
      using 1.prems(3) closest-pair-rec-code.simps by (simp add: True)
    ultimately show ?thesis
      using 1.prems(1) closest-pair-bf-code-eq by simp
  next
    case False

    obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
      using prod.collapse by blast
    define L where L = fst (hd XS_R)
```

obtain YS_L C_{0L} C_{1L} $YS_L{'}$ $\Delta_L{'}$ $C_{0L}{'}$ $C_{1L}{'}$ **where** YSC_{01L}-*def*:
(YS_L, C_{0L}, C_{1L}) = *closest-pair-rec* XS_L
($YS_L{'}$, $\Delta_L{'}$, $C_{0L}{'}$, $C_{1L}{'}$) = *closest-pair-rec-code* XS_L
using *prod.collapse* **by** *metis*
obtain YS_R C_{0R} C_{1R} $YS_R{'}$ $\Delta_R{'}$ $C_{0R}{'}$ $C_{1R}{'}$ **where** YSC_{01R}-*def*:
(YS_R, C_{0R}, C_{1R}) = *closest-pair-rec* XS_R
($YS_R{'}$, $\Delta_R{'}$, $C_{0R}{'}$, $C_{1R}{'}$) = *closest-pair-rec-code* XS_R
using *prod.collapse* **by** *metis*

define YS YS' **where** *YS-def*:
YS = *merge* (λp. *snd p*) YS_L YS_R
YS' = *merge* (λp. *snd p*) $YS_L{'}$ $YS_R{'}$
obtain C_0 C_1 Δ' $C_0{'}$ $C_1{'}$ **where** C_{01}-*def*:
(C_0, C_1) = *combine* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) L YS
(Δ', $C_0{'}$, $C_1{'}$) = *combine-code* ($\Delta_L{'}$, $C_{0L}{'}$, $C_{1L}{'}$) ($\Delta_R{'}$, $C_{0R}{'}$, $C_{1R}{'}$) L YS'
using *prod.collapse* **by** *metis*
note *defs* = XS_{LR}-*def L-def* YSC_{01L}-*def* YSC_{01R}-*def YS-def* C_{01}-*def*

have *1* < *length* XS_L *length* XS_L < *length xs*
using *False 1.prems*(*1*) *defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHL*: C_{0L} = $C_{0L}{'}$ C_{1L} = $C_{1L}{'}$
using *1.IH defs* **by** *metis+*

have *1* < *length* XS_R *length* XS_R < *length xs*
using *False 1.prems*(*1*) *defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHR*: C_{0R} = $C_{0R}{'}$ C_{1R} = $C_{1R}{'}$
using *1.IH defs* **by** *metis+*

have *sorted-snd* YS_L *sorted-snd* YS_R
using *closest-pair-rec-set-length-sorted-snd* YSC_{01L}-*def*(*1*) YSC_{01R}-*def*(*1*)
by *blast+*
hence *sorted-snd YS*
using *sorted-merge sorted-snd-def YS-def* **by** *blast*
moreover have YS = YS'
using *defs* ‹*1* < *length* XS_L› ‹*1* < *length* XS_R› *closest-pair-rec-ys-eq* **by** *blast*
moreover have $\Delta_L{'}$ = *dist-code* $C_{0L}{'}$ $C_{1L}{'}$ $\Delta_R{'}$ = *dist-code* $C_{0R}{'}$ $C_{1R}{'}$
using *defs* ‹*1* < *length* XS_L› ‹*1* < *length* XS_R› *closest-pair-rec-code-dist-eq*
by *blast+*
ultimately have C_0 = $C_0{'}$ C_1 = $C_1{'}$
using *combine-code-eq IHL IHR* C_{01}-*def* **by** *blast+*
moreover have (YS, C_0, C_1) = *closest-pair-rec xs*
using *False closest-pair-rec-simps defs*(*1*,*2*,*3*,*5*,*7*,*9*)
by (*auto simp*: *Let-def split*: *prod.split*)
moreover have (YS', Δ', $C_0{'}$, $C_1{'}$) = *closest-pair-rec-code xs*
using *False closest-pair-rec-code-simps defs*(*1*,*2*,*4*,*6*,*8*,*10*)
by (*auto simp*: *Let-def split*: *prod.split*)
ultimately show *?thesis*
using *1.prems*(*2*,*3*) **by** (*metis Pair-inject*)

qed
qed

declare *closest-pair.simps* [*simp add*]

fun *closest-pair-code* :: *point list* ⇒ (*point* ∗ *point*) **where**
closest-pair-code [] = *undefined*
| *closest-pair-code* [-] = *undefined*
| *closest-pair-code ps* = (*let* (-, -, c_0, c_1) = *closest-pair-rec-code* (*mergesort fst ps*)
in (c_0, c_1))

lemma *closest-pair-code-eq*:
closest-pair ps = *closest-pair-code ps*
proof (*induction ps rule*: *induct-list012*)
case (*3 x y zs*)
obtain *ys* c_0 c_1 *ys'* δ' c_0' c_1' **where** ∗:
(*ys*, c_0, c_1) = *closest-pair-rec* (*mergesort fst* (*x* # *y* # *zs*))
(*ys'*, δ', c_0', c_1') = *closest-pair-rec-code* (*mergesort fst* (*x* # *y* # *zs*))
by (*metis prod-cases3*)
moreover have *1* < *length* (*mergesort fst* (*x* # *y* # *zs*))
using *length-mergesort*[*of fst x* # *y* # *zs*] **by** *simp*
ultimately have c_0 = c_0' c_1 = c_1'
using *closest-pair-rec-code-eq* **by** *blast+*
thus *?case*
using ∗ **by** (*auto split*: *prod.splits*)
qed *auto*

export-code *closest-pair-code* **in** *OCaml*
module-name *Verified*

end

3 Closest Pair Algorithm 2

theory *Closest-Pair-Alternative*
imports *Common*
begin

Formalization of a divide-and-conquer algorithm solving the Closest Pair Problem based on the presentation of Cormen *et al.* [1].

3.1 Functional Correctness Proof

3.1.1 Core Argument

lemma *core-argument*:
assumes *distinct* (*p*$_0$ # *ps*) *sorted-snd* (*p*$_0$ # *ps*) *0* ≤ δ *set* (*p*$_0$ # *ps*) = ps_L ∪ ps_R
assumes ∀ *p* ∈ *set* (*p*$_0$ # *ps*). *l* − δ ≤ *fst p* ∧ *fst p* ≤ *l* + δ

assumes $\forall\, p \in ps_L.\ fst\ p \leq l\ \forall\, p \in ps_R.\ l \leq fst\ p$
assumes *sparse* δ ps_L *sparse* δ ps_R
assumes $p_1 \in set\ ps$ *dist* $p_0\ p_1 < \delta$
shows $p_1 \in set\ (take\ 7\ ps)$
proof −
define PS **where** $PS = p_0 \#\ ps$
define R **where** $R = cbox\ (l - \delta,\ snd\ p_0)\ (l + \delta,\ snd\ p_0 + \delta)$
define RPS **where** $RPS = \{\ p \in set\ PS.\ p \in R\ \}$
define LSQ **where** $LSQ = cbox\ (l - \delta,\ snd\ p_0)\ (l,\ snd\ p_0 + \delta)$
define $LSQPS$ **where** $LSQPS = \{\ p \in ps_L.\ p \in LSQ\ \}$
define RSQ **where** $RSQ = cbox\ (l,\ snd\ p_0)\ (l + \delta,\ snd\ p_0 + \delta)$
define $RSQPS$ **where** $RSQPS = \{\ p \in ps_R.\ p \in RSQ\ \}$
note *defs* = *PS-def R-def RPS-def LSQ-def LSQPS-def RSQ-def RSQPS-def*

have $R = LSQ \cup RSQ$
using *defs cbox-right-un* **by** *auto*
moreover have $\forall\, p \in ps_L.\ p \in RSQ \longrightarrow p \in LSQ$
using *RSQ-def LSQ-def assms*(6) **by** *auto*
moreover have $\forall\, p \in ps_R.\ p \in LSQ \longrightarrow p \in RSQ$
using *RSQ-def LSQ-def assms*(7) **by** *auto*
ultimately have $RPS = LSQPS \cup RSQPS$
using *LSQPS-def RSQPS-def PS-def RPS-def assms*(4) **by** *blast*

have *sparse* δ *LSQPS*
using *assms*(8) *LSQPS-def sparse-def* **by** *simp*
hence *CLSQPS*: *card LSQPS* $\leq$ 4
using *max-points-square*[*of LSQPS l* − δ *snd* p_0 δ] *assms*(3) *LSQ-def LSQPS-def*
by *auto*

have *sparse* δ *RSQPS*
using *assms*(9) *RSQPS-def sparse-def* **by** *simp*
hence *CRSQPS*: *card RSQPS* $\leq$ 4
using *max-points-square*[*of RSQPS l snd* p_0 δ] *assms*(3) *RSQ-def RSQPS-def*
by *auto*

have *CRPS*: *card RPS* $\leq$ 8
using *CLSQPS CRSQPS card-Un-le*[*of LSQPS RSQPS*] ‹*RPS* = *LSQPS* ∪ *RSQPS*› **by** *auto*

have *RPS* $\subseteq$ *set* (*take* 8 *PS*)
proof (*rule ccontr*)
assume ¬ (*RPS* $\subseteq$ *set* (*take* 8 *PS*))
then obtain p **where** ∗: $p \in set\ PS\ p \in RPS\ p \notin set\ (take\ 8\ PS)\ p \in R$
using *RPS-def* **by** *auto*

have $\forall\, p_0 \in set\ (take\ 8\ PS).\ \forall\, p_1 \in set\ (drop\ 8\ PS).\ snd\ p_0 \leq snd\ p_1$
using *sorted-wrt-take-drop*[*of* $\lambda p_0\ p_1.\ snd\ p_0 \leq snd\ p_1$ *PS* 8] *assms*(2) *sorted-snd-def PS-def* **by** *fastforce*
hence $\forall\, p' \in set\ (take\ 8\ PS).\ snd\ p' \leq snd\ p$

using *append-take-drop-id set-append Un-iff* ∗(*1*,*3*) **by** *metis*
moreover have *snd p* ≤ *snd p*$_0$ + *δ*
using ‹*p* ∈ *R*› *R-def mem-cbox-2D* **by** (*metis* (*mono-tags*, *lifting*) *prod.case-eq-if*)
ultimately have ∀ *p* ∈ *set* (*take 8 PS*). *snd p* ≤ *snd p*$_0$ + *δ*
by *fastforce*
moreover have ∀ *p* ∈ *set* (*take 8 PS*). *snd p*$_0$ ≤ *snd p*
using *sorted-wrt-hd-less-take*[*of* λ*p*$_0$ *p*$_1$. *snd p*$_0$ ≤ *snd p*$_1$ *p*$_0$ *ps 8*] *assms*(*2*) *sorted-snd-def PS-def* **by** *fastforce*
moreover have ∀ *p* ∈ *set* (*take 8 PS*). *l* − *δ* ≤ *fst p* ∧ *fst p* ≤ *l* + *δ*
using *assms*(*5*) *PS-def* **by** (*meson in-set-takeD*)
ultimately have ∀ *p* ∈ *set* (*take 8 PS*). *p* ∈ *R*
using *R-def mem-cbox-2D* **by** *fastforce*

hence *set* (*take 8 PS*) ⊆ *RPS*
using *RPS-def set-take-subset* **by** *fastforce*
hence *NINE*: { *p* } ∪ *set* (*take 8 PS*) ⊆ *RPS*
using ∗ **by** *simp*

have *8* ≤ *length PS*
using ∗(*1*,*3*) *nat-le-linear* **by** *fastforce*
hence *length* (*take 8 PS*) = *8*
by *simp*

have *finite* { *p* } *finite* (*set* (*take 8 PS*))
by *simp-all*
hence *card* ({ *p* } ∪ *set* (*take 8 PS*)) = *card* ({ *p* }) + *card* (*set* (*take 8 PS*))
using ∗(*3*) *card-Un-disjoint* **by** *blast*
hence *card* ({ *p* } ∪ *set* (*take 8 PS*)) = *9*
using *assms*(*1*) ‹*length* (*take 8 PS*) = *8*› *distinct-card*[*of take 8 PS*] *distinct-take*[*of PS*] *PS-def* **by** *fastforce*
moreover have *finite RPS*
using *RPS-def* **by** *simp*
ultimately have *9* ≤ *card RPS*
using *NINE card-mono* **by** *metis*
thus *False*
using *CRPS* **by** *simp*
qed

have *dist* (*snd p*$_0$) (*snd p*$_1$) < *δ*
using *assms*(*11*) *dist-snd-le le-less-trans* **by** (*metis* (*no-types*, *lifting*) *prod.case-eq-if snd-conv*)
hence *snd p*$_1$ ≤ *snd p*$_0$ + *δ*
by (*simp add*: *dist-real-def*)
moreover have *l* − *δ* ≤ *fst p*$_1$ *fst p*$_1$ ≤ *l* + *δ*
using *assms*(*5*,*10*) **by** *auto*
moreover have *snd p*$_0$ ≤ *snd p*$_1$
using *sorted-snd-def assms*(*2*,*10*) **by** *auto*
ultimately have *p*$_1$ ∈ *R*
using *mem-cbox-2D*[*of l* − *δ fst p*$_1$ *l* + *δ snd p*$_0$ *snd p*$_1$ *snd p*$_0$ + *δ*] *defs*

by (*simp add*: ‹$l - \delta \leq fst\ p_1$› ‹$snd\ p_0 \leq snd\ p_1$› *prod.case-eq-if*)
moreover have $p_1 \in set\ PS$
using *PS-def assms*(*10*) **by** *simp*
ultimately have $p_1 \in set\ (take\ 8\ PS)$
using *RPS-def* ‹$RPS \subseteq set\ (take\ 8\ PS)$› **by** *auto*
thus *?thesis*
using *assms*(*1*,*10*) *PS-def* **by** *auto*
qed

3.1.2 Combine step

lemma *find-closest-bf-dist-take-7*:
assumes $\exists\, p_1 \in set\ ps.\ dist\ p_0\ p_1 < \delta$
assumes *distinct* $(p_0 \#\ ps)$ *sorted-snd* $(p_0 \#\ ps)$ $0 < length\ ps\ 0 \leq \delta$ $set\ (p_0 \#\ ps) = ps_L \cup ps_R$
assumes $\forall\, p \in set\ (p_0 \#\ ps).\ l - \delta \leq fst\ p \wedge fst\ p \leq l + \delta$
assumes $\forall\, p \in ps_L.\ fst\ p \leq l\ \forall\, p \in ps_R.\ l \leq fst\ p$
assumes *sparse* δ ps_L *sparse* δ ps_R
shows $\forall\, p_1 \in set\ ps.\ dist\ p_0\ (find\text{-}closest\text{-}bf\ p_0\ (take\ 7\ ps)) \leq dist\ p_0\ p_1$
proof −
have $dist\ p_0\ (find\text{-}closest\text{-}bf\ p_0\ ps) < \delta$
using *assms*(*1*) *dual-order.strict-trans2 find-closest-bf-dist* **by** *blast*
moreover have $find\text{-}closest\text{-}bf\ p_0\ ps \in set\ ps$
using *assms*(*4*) *find-closest-bf-set* **by** *blast*
ultimately have $find\text{-}closest\text{-}bf\ p_0\ ps \in set\ (take\ 7\ ps)$
using *core-argument*[*of* p_0 *ps* δ ps_L ps_R *l find-closest-bf* p_0 *ps*] *assms* **by** *blast*
moreover have $\forall\, p_1 \in set\ (take\ 7\ ps).\ dist\ p_0\ (find\text{-}closest\text{-}bf\ p_0\ (take\ 7\ ps)) \leq dist\ p_0\ p_1$
using *find-closest-bf-dist* **by** *blast*
ultimately have $\forall\, p_1 \in set\ ps.\ dist\ p_0\ (find\text{-}closest\text{-}bf\ p_0\ (take\ 7\ ps)) \leq dist\ p_0\ p_1$
using *find-closest-bf-dist order.trans* **by** *blast*
thus *?thesis* **.**
qed

fun *find-closest-pair-tm* :: $(point * point) \Rightarrow point\ list \Rightarrow (point \times point)\ tm$ **where**
find-closest-pair-tm $(c_0,\ c_1)$ [] =*1 return* $(c_0,\ c_1)$
| *find-closest-pair-tm* $(c_0,\ c_1)$ [-] =*1 return* $(c_0,\ c_1)$
| *find-closest-pair-tm* $(c_0,\ c_1)$ $(p_0 \#\ ps)$ =*1* (
do {
ps' <− *take-tm 7 ps*;
p_1 <− *find-closest-bf-tm* p_0 ps';
if dist c_0 $c_1 \leq dist\ p_0\ p_1$ *then*
find-closest-pair-tm $(c_0,\ c_1)$ *ps*
else
find-closest-pair-tm $(p_0,\ p_1)$ *ps*
}
)

```
fun find-closest-pair :: (point * point) ⇒ point list ⇒ (point * point) where
  find-closest-pair (c₀, c₁) [] = (c₀, c₁)
| find-closest-pair (c₀, c₁) [-] = (c₀, c₁)
| find-closest-pair (c₀, c₁) (p₀ # ps) = (
    let p₁ = find-closest-bf p₀ (take 7 ps) in
    if dist c₀ c₁ ≤ dist p₀ p₁ then
      find-closest-pair (c₀, c₁) ps
    else
      find-closest-pair (p₀, p₁) ps
  )

lemma find-closest-pair-eq-val-find-closest-pair-tm:
  val (find-closest-pair-tm (c₀, c₁) ps) = find-closest-pair (c₀, c₁) ps
  by (induction (c₀, c₁) ps arbitrary: c₀ c₁ rule: find-closest-pair.induct)
     (auto simp: Let-def find-closest-bf-eq-val-find-closest-bf-tm take-eq-val-take-tm)

lemma find-closest-pair-set:
  assumes (C₀, C₁) = find-closest-pair (c₀, c₁) ps
  shows (C₀ ∈ set ps ∧ C₁ ∈ set ps) ∨ (C₀ = c₀ ∧ C₁ = c₁)
  using assms
proof (induction (c₀, c₁) ps arbitrary: c₀ c₁ C₀ C₁ rule: find-closest-pair.induct)
  case (3 c₀ c₁ p₀ p₂ ps)
  define p₁ where p₁-def: p₁ = find-closest-bf p₀ (take 7 (p₂ # ps))
  hence A: p₁ ∈ set (p₂ # ps)
    using find-closest-bf-set[of take 7 (p₂ # ps)] in-set-takeD by fastforce
  show ?case
  proof (cases dist c₀ c₁ ≤ dist p₀ p₁)
    case True
    obtain C₀' C₁' where C'-def: (C₀', C₁') = find-closest-pair (c₀, c₁) (p₂ #
ps)
      using prod.collapse by blast
    note defs = p₁-def C'-def
    hence (C₀' ∈ set (p₂ # ps) ∧ C₁' ∈ set (p₂ # ps)) ∨ (C₀' = c₀ ∧ C₁' = c₁)
      using 3.hyps(1) True p₁-def by blast
    moreover have C₀ = C₀' C₁ = C₁'
    using defs True 3.prems apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by auto
  next
    case False
    obtain C₀' C₁' where C'-def: (C₀', C₁') = find-closest-pair (p₀, p₁) (p₂ #
ps)
      using prod.collapse by blast
    note defs = p₁-def C'-def
    hence (C₀' ∈ set (p₂ # ps) ∧ C₁' ∈ set (p₂ # ps)) ∨ (C₀' = p₀ ∧ C₁' = p₁)
      using 3.hyps(2) p₁-def False by blast
    moreover have C₀ = C₀' C₁ = C₁'
    using defs False 3.prems apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
```

```
    using A by auto
  qed
qed auto

lemma find-closest-pair-c0-ne-c1:
  c0 ≠ c1 ⟹ distinct ps ⟹ (C0, C1) = find-closest-pair (c0, c1) ps ⟹ C0 ≠
C1
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case (3 c0 c1 p0 p2 ps)
  define p1 where p1-def: p1 = find-closest-bf p0 (take 7 (p2 # ps))
  hence p1 ∈ set (p2 # ps)
    using find-closest-bf-set[of take 7 (p2 # ps)] in-set-takeD by fastforce
  hence A: p0 ≠ p1
    using 3.prems(1,2) by auto
  show ?case
  proof (cases dist c0 c1 ≤ dist p0 p1)
    case True
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (c0, c1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence C0' ≠ C1'
      using 3.hyps(1) 3.prems(1,2) True p1-def by simp
    moreover have C0 = C0' C1 = C1'
    using defs True 3.prems(3) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by simp
  next
    case False
    obtain C0' C1' where C'-def: (C0', C1') = find-closest-pair (p0, p1) (p2 #
ps)
      using prod.collapse by blast
    note defs = p1-def C'-def
    hence C0' ≠ C1'
      using 3.hyps(2) 3.prems(2) A False p1-def by simp
    moreover have C0 = C0' C1 = C1'
    using defs False 3.prems(3) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by simp
  qed
qed auto

lemma find-closest-pair-dist-mono:
  assumes (C0, C1) = find-closest-pair (c0, c1) ps
  shows dist C0 C1 ≤ dist c0 c1
  using assms
proof (induction (c0, c1) ps arbitrary: c0 c1 C0 C1 rule: find-closest-pair.induct)
  case (3 c0 c1 p0 p2 ps)
  define p1 where p1-def: p1 = find-closest-bf p0 (take 7 (p2 # ps))
```

```
  show ?case
  proof (cases dist c₀ c₁ ≤ dist p₀ p₁)
    case True
    obtain C₀' C₁' where C'-def: (C₀', C₁') = find-closest-pair (c₀, c₁) (p₂ #
ps)
      using prod.collapse by blast
    note defs = p₁-def C'-def
    hence dist C₀' C₁' ≤ dist c₀ c₁
      using 3.hyps(1) True p₁-def by simp
    moreover have C₀ = C₀' C₁ = C₁'
    using defs True 3.prems apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      by simp
  next
    case False
    obtain C₀' C₁' where C'-def: (C₀', C₁') = find-closest-pair (p₀, p₁) (p₂ #
ps)
      using prod.collapse by blast
    note defs = p₁-def C'-def
    hence dist C₀' C₁' ≤ dist p₀ p₁
      using 3.hyps(2) False p₁-def by blast
    moreover have C₀ = C₀' C₁ = C₁'
    using defs False 3.prems(1) apply (auto split: prod.splits) by (metis Pair-inject)+
    ultimately show ?thesis
      using False by simp
  qed
qed auto

lemma find-closest-pair-dist:
  assumes sorted-snd ps distinct ps set ps = psL ∪ psR 0 ≤ δ
  assumes ∀ p ∈ set ps. l − δ ≤ fst p ∧ fst p ≤ l + δ
  assumes ∀ p ∈ psL. fst p ≤ l ∀ p ∈ psR. l ≤ fst p
  assumes sparse δ psL sparse δ psR dist c₀ c₁ ≤ δ
  assumes (C₀, C₁) = find-closest-pair (c₀, c₁) ps
  shows sparse (dist C₀ C₁) (set ps)
  using assms
proof (induction (c₀, c₁) ps arbitrary: c₀ c₁ C₀ C₁ psL psR rule: find-closest-pair.induct)
  case (1 c₀ c₁)
  thus ?case unfolding sparse-def
    by simp
next
  case (2 c₀ c₁ uu)
  thus ?case unfolding sparse-def
    by (metis length-greater-0-conv length-pos-if-in-set set-ConsD)
next
  case (3 c₀ c₁ p₀ p₂ ps)
  define p₁ where p₁-def: p₁ = find-closest-bf p₀ (take 7 (p₂ # ps))
  define PSL where PSL-def: PSL = psL − { p₀ }
  define PSR where PSR-def: PSR = psR − { p₀ }
```

have *assms*: *sorted-snd* $(p_2 \# ps)$ *distinct* $(p_2 \# ps)$ *set* $(p_2 \# ps) = PS_L \cup PS_R$
$\forall\, p \in set\ (p_2 \# ps).\ l - \delta \leq (fst\ p) \wedge (fst\ p) \leq l + \delta$
$\forall\, p \in PS_L.\ fst\ p \leq l\ \forall\, p \in PS_R.\ l \leq fst\ p$
sparse δ PS_L *sparse* δ PS_R
using *3.prems*(1−9) *sparse-def sorted-snd-def* PS_L*-def* PS_R*-def* **by** *auto*

show *?case*
proof *cases*
assume *C1*: $\exists\, p \in set\ (p_2 \# ps).\ dist\ p_0\ p < \delta$
hence *A*: $\forall\, p \in set\ (p_2 \# ps).\ dist\ p_0\ p_1 \leq dist\ p_0\ p$
using p_1*-def find-closest-bf-dist-take-7 3.prems* **by** *blast*
hence *B*: $dist\ p_0\ p_1 < \delta$
using *C1* **by** *auto*
show *?thesis*
proof *cases*
assume *C2*: $dist\ c_0\ c_1 \leq dist\ p_0\ p_1$
obtain $C_0'\ C_1'$ **where** *def*: $(C_0', C_1') = find\text{-}closest\text{-}pair\ (c_0, c_1)\ (p_2 \# ps)$
using *prod.collapse* **by** *blast*
hence *sparse* $(dist\ C_0'\ C_1')\ (set\ (p_2 \# ps))$
using *3.hyps*(1)[*of* p_1 PS_L PS_R] *C2* p_1*-def 3.prems*(4,10) *assms* **by** *blast*
moreover have $dist\ C_0'\ C_1' \leq dist\ c_0\ c_1$
using *def find-closest-pair-dist-mono* **by** *blast*
ultimately have *sparse* $(dist\ C_0'\ C_1')\ (set\ (p_0 \# p_2 \# ps))$
using *A C2 sparse-identity*[*of* $dist\ C_0'\ C_1'\ p_2 \# ps\ p_0$] **by** *fastforce*
moreover have $C_0' = C_0\ C_1' = C_1$
using *def 3.prems*(11) *C2* p_1*-def* **apply** (*auto*) **by** (*metis prod.inject*)+
ultimately show *?thesis*
by *simp*
next
assume *C2*: $\neg\ dist\ c_0\ c_1 \leq dist\ p_0\ p_1$
obtain $C_0'\ C_1'$ **where** *def*: $(C_0', C_1') = find\text{-}closest\text{-}pair\ (p_0, p_1)\ (p_2 \# ps)$
using *prod.collapse* **by** *blast*
hence *sparse* $(dist\ C_0'\ C_1')\ (set\ (p_2 \# ps))$
using *3.hyps*(2)[*of* p_1 PS_L PS_R] *C2* p_1*-def 3.prems*(4) *assms B* **by** *auto*
moreover have $dist\ C_0'\ C_1' \leq dist\ p_0\ p_1$
using *def find-closest-pair-dist-mono* **by** *blast*
ultimately have *sparse* $(dist\ C_0'\ C_1')\ (set\ (p_0 \# p_2 \# ps))$
using *A sparse-identity order-trans* **by** *blast*
moreover have $C_0' = C_0\ C_1' = C_1$
using *def 3.prems*(11) *C2* p_1*-def* **apply** (*auto*) **by** (*metis prod.inject*)+
ultimately show *?thesis*
by *simp*
qed
next
assume *C1*: $\neg\ (\exists\, p \in set\ (p_2 \# ps).\ dist\ p_0\ p < \delta)$
show *?thesis*
proof *cases*

```
      assume C2: dist c0 c1 ≤ dist p0 p1
      obtain C0' C1' where def: (C0', C1') = find-closest-pair (c0, c1) (p2 # ps)
        using prod.collapse by blast
      hence sparse (dist C0' C1') (set (p2 # ps))
        using 3.hyps(1)[of p1 PSL PSR] C2 p1-def 3.prems(4,10) assms by blast
      moreover have dist C0' C1' ≤ dist c0 c1
        using def find-closest-pair-dist-mono by blast
      ultimately have sparse (dist C0' C1') (set (p0 # p2 # ps))
        using 3.prems(10) C1 sparse-identity[of dist C0' C1' p2 # ps p0] by force
      moreover have C0' = C0 C1' = C1
        using def 3.prems(11) C2 p1-def apply (auto) by (metis prod.inject)+
      ultimately show ?thesis
        by simp
    next
      assume C2: ¬ dist c0 c1 ≤ dist p0 p1
      obtain C0' C1' where def: (C0', C1') = find-closest-pair (p0, p1) (p2 # ps)
        using prod.collapse by blast
      hence sparse (dist C0' C1') (set (p2 # ps))
        using 3.hyps(2)[of p1 PSL PSR] C2 p1-def 3.prems(4,10) assms by auto
      moreover have dist C0' C1' ≤ dist p0 p1
        using def find-closest-pair-dist-mono by blast
      ultimately have sparse (dist C0' C1') (set (p0 # p2 # ps))
        using 3.prems(10) C1 C2 sparse-identity[of dist C0' C1' p2 # ps p0] by
force
      moreover have C0' = C0 C1' = C1
        using def 3.prems(11) C2 p1-def apply (auto) by (metis prod.inject)+
      ultimately show ?thesis
        by simp
    qed
  qed
qed

declare find-closest-pair.simps [simp del]

fun combine-tm :: (point × point) ⇒ (point × point) ⇒ int ⇒ point list ⇒ (point
× point) tm where
  combine-tm (p0L, p1L) (p0R, p1R) l ps =1 (
    let (c0, c1) = if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R) in
    do {
      ps' <− filter-tm (λp. dist p (l, snd p) < dist c0 c1) ps;
      find-closest-pair-tm (c0, c1) ps'
    }
  )

fun combine :: (point ∗ point) ⇒ (point ∗ point) ⇒ int ⇒ point list ⇒ (point ∗
point) where
  combine (p0L, p1L) (p0R, p1R) l ps = (
    let (c0, c1) = if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R) in
    let ps' = filter (λp. dist p (l, snd p) < dist c0 c1) ps in
```

find-closest-pair (c_0, c_1) *ps'*
)

lemma *combine-eq-val-combine-tm*:
val (*combine-tm* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
by (*auto simp*: *filter-eq-val-filter-tm find-closest-pair-eq-val-find-closest-pair-tm*)

lemma *combine-set*:
assumes (c_0, c_1) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
shows ($c_0 \in$ *set ps* $\wedge$ $c_1 \in$ *set ps*) $\vee$ ($c_0 = p_{0L} \wedge c_1 = p_{1L}$) $\vee$ ($c_0 = p_{0R} \wedge c_1 = p_{1R}$)
proof −
obtain C_0' C_1' **where** *C'-def*: (C_0', C_1') = (*if dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}))
by *metis*
define *ps'* **where** *ps'-def*: *ps'* = *filter* (λ*p*. *dist p* (*l*, *snd p*) < *dist* C_0' C_1') *ps*
obtain C_0 C_1 **where** *C-def*: (C_0, C_1) = *find-closest-pair* (C_0', C_1') *ps'*
using *prod.collapse* **by** *blast*
note *defs* = *C'-def ps'-def C-def*
have ($C_0 \in$ *set ps'* $\wedge$ $C_1 \in$ *set ps'*) $\vee$ ($C_0 = C_0' \wedge C_1 = C_1'$)
using *C-def find-closest-pair-set* **by** *blast+*
hence ($C_0 \in$ *set ps* $\wedge$ $C_1 \in$ *set ps*)$\vee$ ($C_0 = C_0' \wedge C_1 = C_1'$)
using *ps'-def* **by** *auto*
moreover have $C_0 = c_0$ $C_1 = c_1$
using *assms defs* **apply** (*auto split*: *if-splits prod.splits*) **by** (*metis Pair-inject*)+
ultimately show *?thesis*
using *C'-def* **by** (*auto split*: *if-splits*)
qed

lemma *combine-c0-ne-c1*:
assumes $p_{0L} \neq p_{1L}$ $p_{0R} \neq p_{1R}$ *distinct ps*
assumes (c_0, c_1) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
shows $c_0 \neq c_1$
proof −
obtain C_0' C_1' **where** *C'-def*: (C_0', C_1') = (*if dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}))
by *metis*
define *ps'* **where** *ps'-def*: *ps'* = *filter* (λ*p*. *dist p* (*l*, *snd p*) < *dist* C_0' C_1') *ps*
obtain C_0 C_1 **where** *C-def*: (C_0, C_1) = *find-closest-pair* (C_0', C_1') *ps'*
using *prod.collapse* **by** *blast*
note *defs* = *C'-def ps'-def C-def*
have $C_0 \neq C_1$
using *defs find-closest-pair-c0-ne-c1*[*of* C_0' C_1' *ps'*] *assms* **by** (*auto split*: *if-splits*)
moreover have $C_0 = c_0$ $C_1 = c_1$
using *assms defs* **apply** (*auto split*: *if-splits prod.splits*) **by** (*metis Pair-inject*)+
ultimately show *?thesis*
by *blast*

qed

lemma *combine-dist*:
assumes *distinct ps sorted-snd ps set ps* = $ps_L \cup ps_R$
assumes $\forall\, p \in ps_L$. *fst p* $\leq l\ \forall\, p \in ps_R$. $l \leq$ *fst p*
assumes *sparse* (*dist* p_{0L} p_{1L}) ps_L *sparse* (*dist* p_{0R} p_{1R}) ps_R
assumes (c_0, c_1) = *combine* (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) *l ps*
shows *sparse* (*dist* c_0 c_1) (*set ps*)
proof −
obtain $C_0{'}$ $C_1{'}$ **where** *C'-def*: ($C_0{'}$, $C_1{'}$) = (*if dist* p_{0L} p_{1L} < *dist* p_{0R} p_{1R} *then* (p_{0L}, p_{1L}) *else* (p_{0R}, p_{1R}))
by *metis*
define *ps'* **where** *ps'-def*: *ps'* = *filter* (λ*p*. *dist p* (*l*, *snd p*) < *dist* $C_0{'}$ $C_1{'}$) *ps*
define PS_L **where** PS_L*-def*: PS_L = { $p \in ps_L$. *dist p* (*l*, *snd p*) < *dist* $C_0{'}$ $C_1{'}$ }
define PS_R **where** PS_R*-def*: PS_R = { $p \in ps_R$. *dist p* (*l*, *snd p*) < *dist* $C_0{'}$ $C_1{'}$ }
obtain C_0 C_1 **where** *C-def*: (C_0, C_1) = *find-closest-pair* ($C_0{'}$, $C_1{'}$) *ps'*
using *prod.collapse* **by** *blast*
note *defs* = *C'-def ps'-def* PS_L*-def* PS_R*-def C-def*
have *EQ*: $C_0 = c_0$ $C_1 = c_1$
using *defs assms*(8) **apply** (*auto split*: *if-splits prod.splits*) **by** (*metis Pair-inject*)+
have *ps'*: *ps'* = *filter* (λ*p*. *l* − *dist* $C_0{'}$ $C_1{'}$ < *fst p* ∧ *fst p* < *l* + *dist* $C_0{'}$ $C_1{'}$) *ps*
using *ps'-def dist-transform* **by** *simp*
have ps_L: *sparse* (*dist* $C_0{'}$ $C_1{'}$) ps_L
using *assms*(6,8) *C'-def sparse-def* **apply** (*auto split*: *if-splits*) **by** *force*+
hence PS_L: *sparse* (*dist* $C_0{'}$ $C_1{'}$) PS_L
using PS_L*-def* **by** (*simp add*: *sparse-def*)
have ps_R: *sparse* (*dist* $C_0{'}$ $C_1{'}$) ps_R
using *assms*(5,7) *C'-def sparse-def* **apply** (*auto split*: *if-splits*) **by** *force*+
hence PS_R: *sparse* (*dist* $C_0{'}$ $C_1{'}$) PS_R
using PS_R*-def* **by** (*simp add*: *sparse-def*)
have *sorted-snd ps'*
using *ps'-def assms*(2) *sorted-snd-def sorted-wrt-filter* **by** *blast*
moreover have *distinct ps'*
using *ps'-def assms*(1) *distinct-filter* **by** *blast*
moreover have *set ps'* = $PS_L \cup PS_R$
using *ps'-def* PS_L*-def* PS_R*-def assms*(3) *filter-Un* **by** *auto*
moreover have 0 ≤ *dist* $C_0{'}$ $C_1{'}$
by *simp*
moreover have ∀ *p* ∈ *set ps'*. *l* − *dist* $C_0{'}$ $C_1{'}$ ≤ *fst p* ∧ *fst p* ≤ *l* + *dist* $C_0{'}$ $C_1{'}$
using *ps'* **by** *simp*
ultimately have ∗: *sparse* (*dist* C_0 C_1) (*set ps'*)
using *find-closest-pair-dist*[*of ps'* PS_L PS_R *dist* $C_0{'}$ $C_1{'}$ *l* $C_0{'}$ $C_1{'}$] *C-def* PS_L PS_R
by (*simp add*: PS_L*-def* PS_R*-def assms*(4,5))
have ∀ $p_0 \in$ *set ps*. ∀ $p_1 \in$ *set ps*. $p_0 \neq p_1 \wedge$ *dist* p_0 p_1 < *dist* $C_0{'}$ $C_1{'} \longrightarrow p_0 \in$

set ps' ∧ p_1 ∈ *set ps'*
using *set-band-filter ps' ps$_L$ ps$_R$ assms(3,4,5)* **by** *blast*
moreover have *dist* C_0 C_1 ≤ *dist* C_0' C_1'
using *C-def find-closest-pair-dist-mono* **by** *blast*
ultimately have ∀ p_0 ∈ *set ps*. ∀ p_1 ∈ *set ps*. p_0 ≠ p_1 ∧ *dist* p_0 p_1 < *dist* C_0 C_1 ⟶ p_0 ∈ *set ps'* ∧ p_1 ∈ *set ps'*
by *simp*
hence *sparse* (*dist* C_0 C_1) (*set ps*)
using *sparse-def* ∗ **by** (*meson not-less*)
thus *?thesis*
using *EQ* **by** *blast*
qed

declare *combine.simps* [*simp del*]
declare *combine-tm.simps* [*simp del*]

3.1.3 Divide and Conquer Algorithm

declare *split-at-take-drop-conv* [*simp add*]

function *closest-pair-rec-tm* :: *point list* ⇒ (*point list* × *point* × *point*) *tm* **where**
closest-pair-rec-tm xs =1 (
do {
n <− *length-tm xs*;
if n ≤ *3 then*
do {
ys <− *mergesort-tm snd xs*;
p <− *closest-pair-bf-tm xs*;
return (*ys*, *p*)
}
else
do {
(*xs$_L$*, *xs$_R$*) <− *split-at-tm* (*n div 2*) *xs*;
(*ys$_L$*, *p$_{0L}$*, *p$_{1L}$*) <− *closest-pair-rec-tm xs$_L$*;
(*ys$_R$*, *p$_{0R}$*, *p$_{1R}$*) <− *closest-pair-rec-tm xs$_R$*;
ys <− *merge-tm snd ys$_L$ ys$_R$*;
(*p*$_0$, *p*$_1$) <− *combine-tm* (*p$_{0L}$*, *p$_{1L}$*) (*p$_{0R}$*, *p$_{1R}$*) (*fst* (*hd xs$_R$*)) *ys*;
return (*ys*, *p*$_0$, *p*$_1$)
}
}
)
by *pat-completeness auto*
termination *closest-pair-rec-tm*
by (*relation Wellfounded.measure* (λ*xs*. *length xs*))
(*auto simp add*: *length-eq-val-length-tm split-at-eq-val-split-at-tm*)

function *closest-pair-rec* :: *point list* ⇒ (*point list* ∗ *point* ∗ *point*) **where**
closest-pair-rec xs = (
let n = *length xs in*

```
    if n ≤ 3 then
      (mergesort snd xs, closest-pair-bf xs)
    else
      let (xs_L, xs_R) = split-at (n div 2) xs in
      let (ys_L, p_0L, p_1L) = closest-pair-rec xs_L in
      let (ys_R, p_0R, p_1R) = closest-pair-rec xs_R in
      let ys = merge snd ys_L ys_R in
      (ys, combine (p_0L, p_1L) (p_0R, p_1R) (fst (hd xs_R)) ys)
  )
  by pat-completeness auto
termination closest-pair-rec
  by (relation Wellfounded.measure (λxs. length xs))
    (auto simp: Let-def)

declare split-at-take-drop-conv [simp del]

lemma closest-pair-rec-simps:
  assumes n = length xs ¬ (n ≤ 3)
  shows closest-pair-rec xs = (
    let (xs_L, xs_R) = split-at (n div 2) xs in
    let (ys_L, p_0L, p_1L) = closest-pair-rec xs_L in
    let (ys_R, p_0R, p_1R) = closest-pair-rec xs_R in
    let ys = merge snd ys_L ys_R in
    (ys, combine (p_0L, p_1L) (p_0R, p_1R) (fst (hd xs_R)) ys)
  )
  using assms by (auto simp: Let-def)

declare closest-pair-rec.simps [simp del]

lemma closest-pair-rec-eq-val-closest-pair-rec-tm:
  val (closest-pair-rec-tm xs) = closest-pair-rec xs
proof (induction rule: length-induct)
  case (1 xs)
  define n where n = length xs
  obtain xs_L xs_R where xs-def: (xs_L, xs_R) = split-at (n div 2) xs
    by (metis surj-pair)
  note defs = n-def xs-def
  show ?case
  proof cases
    assume n ≤ 3
    then show ?thesis
      using defs
      by (auto simp: length-eq-val-length-tm mergesort-eq-val-mergesort-tm
                closest-pair-bf-eq-val-closest-pair-bf-tm closest-pair-rec.simps)
  next
    assume asm: ¬ n ≤ 3
    have length xs_L < length xs length xs_R < length xs
      using asm defs by (auto simp: split-at-take-drop-conv)
    hence val (closest-pair-rec-tm xs_L) = closest-pair-rec xs_L
```

```
        val (closest-pair-rec-tm xs_R) = closest-pair-rec xs_R
      using 1.IH by blast+
    thus ?thesis
      using asm defs
      apply (subst closest-pair-rec.simps, subst closest-pair-rec-tm.simps)
      by (auto simp del: closest-pair-rec-tm.simps
            simp add: Let-def length-eq-val-length-tm merge-eq-val-merge-tm
                  split-at-eq-val-split-at-tm combine-eq-val-combine-tm
            split: prod.split)
  qed
qed

lemma closest-pair-rec-set-length-sorted-snd:
  assumes (ys, p) = closest-pair-rec xs
  shows set ys = set xs ∧ length ys = length xs ∧ sorted-snd ys
  using assms
proof (induction xs arbitrary: ys p rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    thus ?thesis using 1.prems sorted-snd-def
      by (auto simp: mergesort closest-pair-rec.simps)
  next
    case False

    obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
      using prod.collapse by blast
    define L where L = fst (hd XS_R)
    obtain YS_L P_L where YSP_L-def: (YS_L, P_L) = closest-pair-rec XS_L
      using prod.collapse by blast
    obtain YS_R P_R where YSP_R-def: (YS_R, P_R) = closest-pair-rec XS_R
      using prod.collapse by blast
    define YS where YS = merge (λp. snd p) YS_L YS_R
    define P where P = combine P_L P_R L YS
    note defs = XS_LR-def L-def YSP_L-def YSP_R-def YS-def P-def

    have length XS_L < length xs length XS_R < length xs
      using False defs by (auto simp: split-at-take-drop-conv)
    hence IH: set XS_L = set YS_L set XS_R = set YS_R
          length XS_L = length YS_L length XS_R = length YS_R
          sorted-snd YS_L sorted-snd YS_R
      using 1.IH defs by metis+

    have set xs = set XS_L ∪ set XS_R
      using defs by (auto simp: set-take-drop split-at-take-drop-conv)
    hence SET: set xs = set YS
      using set-merge IH(1,2) defs by fast
```

```
    have length xs = length XS_L + length XS_R
      using defs by (auto simp: split-at-take-drop-conv)
    hence LENGTH: length xs = length YS
      using IH(3,4) length-merge defs by metis

    have SORTED: sorted-snd YS
      using IH(5,6) by (simp add: defs sorted-snd-def sorted-merge)

    have (YS, P) = closest-pair-rec xs
     using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
    hence (ys, p) = (YS, P)
      using 1.prems by argo
    thus ?thesis
      using SET LENGTH SORTED by simp
  qed
qed

lemma closest-pair-rec-distinct:
  assumes distinct xs (ys, p) = closest-pair-rec xs
  shows distinct ys
  using assms
proof (induction xs arbitrary: ys p rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    thus ?thesis using 1.prems
      by (auto simp: mergesort closest-pair-rec.simps)
  next
    case False

    obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
      using prod.collapse by blast
    define L where L = fst (hd XS_R)
    obtain YS_L P_L where YSP_L-def: (YS_L, P_L) = closest-pair-rec XS_L
      using prod.collapse by blast
    obtain YS_R P_R where YSP_R-def: (YS_R, P_R) = closest-pair-rec XS_R
      using prod.collapse by blast
    define YS where YS = merge (λp. snd p) YS_L YS_R
    define P where P = combine P_L P_R L YS
    note defs = XS_LR-def L-def YSP_L-def YSP_R-def YS-def P-def

    have length XS_L < length xs length XS_R < length xs
      using False defs by (auto simp: split-at-take-drop-conv)
    moreover have distinct XS_L distinct XS_R
      using 1.prems(1) defs by (auto simp: split-at-take-drop-conv)
    ultimately have IH: distinct YS_L distinct YS_R
```

```
    using 1.IH defs by blast+

  have set XS_L ∩ set XS_R = {}
  using 1.prems(1) defs by (auto simp: split-at-take-drop-conv set-take-disj-set-drop-if-distinct)
  moreover have set XS_L = set YS_L set XS_R = set YS_R
    using closest-pair-rec-set-length-sorted-snd defs by blast+
  ultimately have set YS_L ∩ set YS_R = {}
    by blast
  hence DISTINCT: distinct YS
    using distinct-merge IH defs by blast

  have (YS, P) = closest-pair-rec xs
   using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
  hence (ys, p) = (YS, P)
    using 1.prems by argo
  thus ?thesis
    using DISTINCT by blast
 qed
qed

lemma closest-pair-rec-c0-c1:
 assumes 1 < length xs distinct xs (ys, c_0, c_1) = closest-pair-rec xs
 shows c_0 ∈ set xs ∧ c_1 ∈ set xs ∧ c_0 ≠ c_1
 using assms
proof (induction xs arbitrary: ys c_0 c_1 rule: length-induct)
 case (1 xs)
 let ?n = length xs
 show ?case
 proof (cases ?n ≤ 3)
  case True
  hence (c_0, c_1) = closest-pair-bf xs
    using 1.prems(3) closest-pair-rec.simps by simp
  thus ?thesis
    using 1.prems(1,2) closest-pair-bf-c0-c1 by simp
 next
  case False

  obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
    using prod.collapse by blast
  define L where L = fst (hd XS_R)

  obtain YS_L C_0L C_1L where YSC_01L-def: (YS_L, C_0L, C_1L) = closest-pair-rec
XS_L
    using prod.collapse by metis
  obtain YS_R C_0R C_1R where YSC_01R-def: (YS_R, C_0R, C_1R) = closest-pair-rec
XS_R
    using prod.collapse by metis

  define YS where YS = merge (λp. snd p) YS_L YS_R
```

```
    obtain C0 C1 where C01-def: (C0, C1) = combine (C0L, C1L) (C0R, C1R)
L YS
      using prod.collapse by metis
    note defs = XSLR-def L-def YSC01L-def YSC01R-def YS-def C01-def

    have 1 < length XSL length XSL < length xs distinct XSL
      using False 1.prems(2) defs by (auto simp: split-at-take-drop-conv)
    hence C0L ∈ set XSL C1L ∈ set XSL and IHL1: C0L ≠ C1L
      using 1.IH defs by metis+
    hence IHL2: C0L ∈ set xs C1L ∈ set xs
      using split-at-take-drop-conv in-set-takeD fst-conv defs by metis+

    have 1 < length XSR length XSR < length xs distinct XSR
      using False 1.prems(2) defs by (auto simp: split-at-take-drop-conv)
    hence C0R ∈ set XSR C1R ∈ set XSR and IHR1: C0R ≠ C1R
      using 1.IH defs by metis+
    hence IHR2: C0R ∈ set xs C1R ∈ set xs
      using split-at-take-drop-conv in-set-dropD snd-conv defs by metis+

    have ∗: (YS, C0, C1) = closest-pair-rec xs
     using False closest-pair-rec-simps defs by (auto simp: Let-def split: prod.split)
    have YS: set xs = set YS distinct YS
     using 1.prems(2) closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct
∗ by blast+

    have C0 ∈ set xs C1 ∈ set xs
      using combine-set IHL2 IHR2 YS defs by blast+
    moreover have C0 ≠ C1
      using combine-c0-ne-c1 IHL1(1) IHR1(1) YS defs by blast
    ultimately show ?thesis
      using 1.prems(3) ∗ by (metis Pair-inject)
  qed
qed

lemma closest-pair-rec-dist:
  assumes 1 < length xs distinct xs sorted-fst xs (ys, c0, c1) = closest-pair-rec xs
  shows sparse (dist c0 c1) (set xs)
  using assms
proof (induction xs arbitrary: ys c0 c1 rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence (c0, c1) = closest-pair-bf xs
      using 1.prems(4) closest-pair-rec.simps by simp
    thus ?thesis
      using 1.prems(1,4) closest-pair-bf-dist by metis
  next
```

case *False*

obtain XS_L XS_R **where** XS_{LR}*-def*: $(XS_L, XS_R) =$ *split-at* (*?n div 2*) *xs*
using *prod.collapse* **by** *blast*
define L **where** $L =$ *fst* (*hd* XS_R)

obtain YS_L C_{0L} C_{1L} **where** YSC_{01L}*-def*: $(YS_L, C_{0L}, C_{1L}) =$ *closest-pair-rec* XS_L
using *prod.collapse* **by** *metis*
obtain YS_R C_{0R} C_{1R} **where** YSC_{01R}*-def*: $(YS_R, C_{0R}, C_{1R}) =$ *closest-pair-rec* XS_R
using *prod.collapse* **by** *metis*

define *YS* **where** *YS* = *merge* (*λp. snd p*) YS_L YS_R
obtain C_0 C_1 **where** C_{01}*-def*: $(C_0, C_1) =$ *combine* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) *L YS*
using *prod.collapse* **by** *metis*
note *defs* = XS_{LR}*-def L-def* YSC_{01L}*-def* YSC_{01R}*-def YS-def* C_{01}*-def*

have *XSLR*: $XS_L =$ *take* (*?n div 2*) *xs* $XS_R =$ *drop* (*?n div 2*) *xs*
using *defs* **by** (*auto simp*: *split-at-take-drop-conv*)

have *1* < *length* XS_L *length* $XS_L <$ *length xs*
using *False XSLR* **by** *simp-all*
moreover have *distinct* XS_L *sorted-fst* XS_L
using *1.prems*(*2*,*3*) *XSLR* **by** (*auto simp*: *sorted-fst-def sorted-wrt-take*)
ultimately have *L*: *sparse* (*dist* C_{0L} C_{1L}) (*set* XS_L)
set $XS_L =$ *set* YS_L
using *1 closest-pair-rec-set-length-sorted-snd closest-pair-rec-c0-c1* YSC_{01L}*-def* **by** *blast+*
hence *IHL*: *sparse* (*dist* C_{0L} C_{1L}) (*set* YS_L)
by *argo*

have *1* < *length* XS_R *length* $XS_R <$ *length xs*
using *False XSLR* **by** *simp-all*
moreover have *distinct* XS_R *sorted-fst* XS_R
using *1.prems*(*2*,*3*) *XSLR* **by** (*auto simp*: *sorted-fst-def sorted-wrt-drop*)
ultimately have *R*: *sparse* (*dist* C_{0R} C_{1R}) (*set* XS_R)
set $XS_R =$ *set* YS_R
using *1 closest-pair-rec-set-length-sorted-snd closest-pair-rec-c0-c1* YSC_{01R}*-def* **by** *blast+*
hence *IHR*: *sparse* (*dist* C_{0R} C_{1R}) (*set* YS_R)
by *argo*

have ∗: $(YS, C_0, C_1) =$ *closest-pair-rec xs*
using *False closest-pair-rec-simps defs* **by** (*auto simp*: *Let-def split*: *prod.split*)

have *set xs* = *set YS distinct YS sorted-snd YS*
using *1.prems*(*2*) *closest-pair-rec-set-length-sorted-snd closest-pair-rec-distinct*

```
* by blast+
    moreover have ∀ p ∈ set YS_L. fst p ≤ L
     using False 1.prems(3) XSLR L-def L(2) sorted-fst-take-less-hd-drop by simp
    moreover have ∀ p ∈ set YS_R. L ≤ fst p
     using False 1.prems(3) XSLR L-def R(2) sorted-fst-hd-drop-less-drop by simp
    moreover have set YS = set YS_L ∪ set YS_R
      using set-merge defs by fast
    moreover have (C_0, C_1) = combine (C_0L, C_1L) (C_0R, C_1R) L YS
      by (auto simp add: defs)
    ultimately have sparse (dist C_0 C_1) (set xs)
      using combine-dist IHL IHR by auto
    moreover have (YS, C_0, C_1) = (ys, c_0, c_1)
      using 1.prems(4) * by simp
    ultimately show ?thesis
      by blast
  qed
qed

fun closest-pair-tm :: point list ⇒ (point * point) tm where
  closest-pair-tm [] =1 return undefined
| closest-pair-tm [-] =1 return undefined
| closest-pair-tm ps =1 (
    do {
      xs <- mergesort-tm fst ps;
      (-, p) <- closest-pair-rec-tm xs;
      return p
    }
  )

fun closest-pair :: point list ⇒ (point * point) where
  closest-pair [] = undefined
| closest-pair [-] = undefined
| closest-pair ps = (let (-, c_0, c_1) = closest-pair-rec (mergesort fst ps) in (c_0, c_1))

lemma closest-pair-eq-val-closest-pair-tm:
  val (closest-pair-tm ps) = closest-pair ps
  by (induction ps rule: induct-list012)
     (auto simp del: closest-pair-rec-tm.simps mergesort-tm.simps
       simp add: closest-pair-rec-eq-val-closest-pair-rec-tm mergesort-eq-val-mergesort-tm
         split: prod.split)

lemma closest-pair-simps:
  1 < length ps ⟹ closest-pair ps = (let (-, c_0, c_1) = closest-pair-rec (mergesort
fst ps) in (c_0, c_1))
  by (induction ps rule: induct-list012) auto

declare closest-pair.simps [simp del]

theorem closest-pair-c0-c1:
```

assumes *1 < length ps distinct ps* $(c_0, c_1) =$ *closest-pair ps*
shows $c_0 \in$ *set ps* $c_1 \in$ *set ps* $c_0 \neq c_1$
using *assms closest-pair-rec-c0-c1*[*of mergesort fst ps*]
by (*auto simp*: *mergesort closest-pair-simps split*: *prod.splits*)

theorem *closest-pair-dist*:
assumes *1 < length ps distinct ps* $(c_0, c_1) =$ *closest-pair ps*
shows *sparse* (*dist* c_0 c_1) (*set ps*)
using *assms closest-pair-rec-dist*[*of mergesort fst ps*] *closest-pair-rec-c0-c1*[*of mergesort fst ps*]
by (*auto simp*: *sorted-fst-def mergesort closest-pair-simps split*: *prod.splits*)

3.2 Time Complexity Proof

3.2.1 Combine Step

lemma *time-find-closest-pair-tm*:
time (*find-closest-pair-tm* (c_0, c_1) *ps*) $\leq$ *17* $*$ *length ps* + *1*
proof (*induction ps rule*: *find-closest-pair-tm.induct*)
case (*3* c_0 c_1 p_0 p_2 *ps*)
let *?ps* = p_2 # *ps*
let *?*p_1 = *val* (*find-closest-bf-tm* p_0 (*val* (*take-tm 7 ?ps*)))
have $*$: *length* (*val* (*take-tm 7 ?ps*)) $\leq$ *7*
by (*subst take-eq-val-take-tm*, *simp*)
show *?case*
proof *cases*
assume *C1*: *dist* c_0 $c_1 \leq$ *dist* p_0 *?*p_1
hence *time* (*find-closest-pair-tm* (c_0, c_1) (p_0 # *?ps*)) = *1* + *time* (*take-tm 7 ?ps*) +
time (*find-closest-bf-tm* p_0 (*val* (*take-tm 7 ?ps*))) + *time* (*find-closest-pair-tm* (c_0, c_1) *?ps*)
by (*auto simp*: *time-simps*)
also have ... $\leq$ *17* + *time* (*find-closest-pair-tm* (c_0, c_1) *?ps*)
using *time-take-tm*[*of 7 ?ps*] *time-find-closest-bf-tm*[*of* p_0 *val* (*take-tm 7 ?ps*)] $*$ **by** *auto*
also have ... $\leq$ *17* + *17* $*$ (*length ?ps*) + *1*
using *3.IH*(*1*) *C1* **by** *simp*
also have ... = *17* $*$ *length* (p_0 # *?ps*) + *1*
by *simp*
finally show *?thesis* **.**
next
assume *C1*: ¬ *dist* c_0 $c_1 \leq$ *dist* p_0 *?*p_1
hence *time* (*find-closest-pair-tm* (c_0, c_1) (p_0 # *?ps*)) = *1* + *time* (*take-tm 7 ?ps*) +
time (*find-closest-bf-tm* p_0 (*val* (*take-tm 7 ?ps*))) + *time* (*find-closest-pair-tm* (p_0, *?*p_1) *?ps*)
by (*auto simp*: *time-simps*)
also have ... $\leq$ *17* + *time* (*find-closest-pair-tm* (p_0, *?*p_1) *?ps*)
using *time-take-tm*[*of 7 ?ps*] *time-find-closest-bf-tm*[*of* p_0 *val* (*take-tm 7 ?ps*)] $*$ **by** *auto*

```
    also have ... ≤ 17 + 17 * (length ?ps) + 1
      using 3.IH(2) C1 by simp
    also have ... = 17 * length (p₀ # ?ps) + 1
      by simp
    finally show ?thesis .
  qed
qed (auto simp: time-simps)

lemma time-combine-tm:
  fixes ps :: point list
  shows time (combine-tm (p0L, p1L) (p0R, p1R) l ps) ≤ 3 + 18 * length ps
proof −
  obtain c0 c1 where c-def:
    (c0, c1) = (if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R)) by
metis
  let ?P = (λp. dist p (l, snd p) < dist c0 c1)
  define ps' where ps' = val (filter-tm ?P ps)
  note defs = c-def ps'-def
  hence time (combine-tm (p0L, p1L) (p0R, p1R) l ps) = 1 + time (filter-tm ?P
ps) +
        time (find-closest-pair-tm (c0, c1) ps')
    by (auto simp: combine-tm.simps Let-def time-simps split: prod.split)
  also have ... = 2 + length ps + time (find-closest-pair-tm (c0, c1) ps')
    using time-filter-tm by auto
  also have ... ≤ 3 + length ps + 17 * length ps'
    using defs time-find-closest-pair-tm by simp
  also have ... ≤ 3 + 18 * length ps
     unfolding ps'-def by (subst filter-eq-val-filter-tm, simp)
  finally show ?thesis
    by blast
qed
```

3.2.2 Divide and Conquer Algorithm

```
lemma time-closest-pair-rec-tm-simps-1:
  assumes length xs ≤ 3
 shows time (closest-pair-rec-tm xs) = 1 + time (length-tm xs) + time (mergesort-tm
snd xs) + time (closest-pair-bf-tm xs)
  using assms by  (auto simp: time-simps length-eq-val-length-tm)

lemma time-closest-pair-rec-tm-simps-2:
  assumes ¬ (length xs ≤ 3)
  shows time (closest-pair-rec-tm xs) = 1 + (
    let (xsL, xsR) = val (split-at-tm (length xs div 2) xs) in
    let (ysL, pL) = val (closest-pair-rec-tm xsL) in
    let (ysR, pR) = val (closest-pair-rec-tm xsR) in
    let ys = val (merge-tm (λp. snd p) ysL ysR) in
   time (length-tm xs) + time (split-at-tm (length xs div 2) xs) + time (closest-pair-rec-tm
xsL) +
```

```
    time (closest-pair-rec-tm xs_R) + time (merge-tm (λp. snd p) ys_L ys_R) + time
(combine-tm p_L p_R (fst (hd xs_R)) ys)
  )
  using assms
  apply (subst closest-pair-rec-tm.simps)
  by (auto simp del: closest-pair-rec-tm.simps
          simp add: time-simps length-eq-val-length-tm
            split: prod.split)

function closest-pair-recurrence :: nat ⇒ real where
  n ≤ 3 ⟹ closest-pair-recurrence n = 3 + n + mergesort-recurrence n + n * n
| 3 < n ⟹ closest-pair-recurrence n = 7 + 21 * n + closest-pair-recurrence (nat
⌊real n / 2⌋) +
    closest-pair-recurrence (nat ⌈real n / 2⌉)
  by force simp-all
termination by akra-bazzi-termination simp-all

lemma closest-pair-recurrence-nonneg[simp]:
  0 ≤ closest-pair-recurrence n
  by (induction n rule: closest-pair-recurrence.induct) auto

lemma time-closest-pair-rec-conv-closest-pair-recurrence:
  time (closest-pair-rec-tm ps) ≤ closest-pair-recurrence (length ps)
proof (induction ps rule: length-induct)
  case (1 ps)
  let ?n = length ps
  show ?case
  proof (cases ?n ≤ 3)
    case True
   hence time (closest-pair-rec-tm ps) = 1 + time (length-tm ps) + time (mergesort-tm
snd ps) + time (closest-pair-bf-tm ps)
      using time-closest-pair-rec-tm-simps-1 by simp
    moreover have closest-pair-recurrence ?n = 3 + ?n + mergesort-recurrence
?n + ?n * ?n
      using True by simp
    moreover have time (length-tm ps) ≤ 1 + ?n time (mergesort-tm snd ps) ≤
mergesort-recurrence ?n
                time (closest-pair-bf-tm ps) ≤ 1 + ?n * ?n
      using time-length-tm[of ps] time-mergesort-conv-mergesort-recurrence[of snd
ps] time-closest-pair-bf-tm[of ps] by auto
    ultimately show ?thesis
      by linarith
  next
    case False

    obtain XS_L XS_R where XS-def: (XS_L, XS_R) = val (split-at-tm (?n div 2)
ps)
      using prod.collapse by blast
   obtain YS_L C_0L C_1L where CP_L-def: (YS_L, C_0L, C_1L) = val (closest-pair-rec-tm
```

XS_L)
using *prod.collapse* **by** *metis*
obtain YS_R C_{0R} C_{1R} **where** CP_R*-def*: (YS_R, C_{0R}, C_{1R}) = *val* (*closest-pair-rec-tm* XS_R)
using *prod.collapse* **by** *metis*
define *YS* **where** *YS* = *val* (*merge-tm* (λ*p*. *snd p*) YS_L YS_R)
obtain C_0 C_1 **where** C_{01}*-def*: (C_0, C_1) = *val* (*combine-tm* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (*fst* (*hd* XS_R)) *YS*)
using *prod.collapse* **by** *metis*
note *defs* = *XS-def* CP_L*-def* CP_R*-def YS-def* C_{01}*-def*

have *XSLR*: XS_L = *take* (*?n div 2*) *ps* XS_R = *drop* (*?n div 2*) *ps*
using *defs* **by** (*auto simp*: *split-at-take-drop-conv split-at-eq-val-split-at-tm*)
hence *length* XS_L = *?n div 2 length* XS_R = *?n* − *?n div 2*
by *simp-all*
hence ∗: (*nat* ⌊*real ?n* / *2*⌋) = *length* XS_L (*nat* ⌈*real ?n* / *2*⌉) = *length* XS_R
by *linarith*+
have *length* XS_L = *length* YS_L *length* XS_R = *length* YS_R
using *defs closest-pair-rec-set-length-sorted-snd closest-pair-rec-eq-val-closest-pair-rec-tm* **by** *metis*+
hence *L*: *?n* = *length* YS_L + *length* YS_R
using *defs XSLR* **by** *fastforce*

have *1* < *length* XS_L *length* XS_L < *length ps*
using *False XSLR* **by** *simp-all*
hence *time* (*closest-pair-rec-tm* XS_L) ≤ *closest-pair-recurrence* (*length* XS_L)
using *1.IH* **by** *simp*
hence *IHL*: *time* (*closest-pair-rec-tm* XS_L) ≤ *closest-pair-recurrence* (*nat* ⌊*real ?n* / *2*⌋)
using ∗ **by** *simp*

have *1* < *length* XS_R *length* XS_R < *length ps*
using *False XSLR* **by** *simp-all*
hence *time* (*closest-pair-rec-tm* XS_R) ≤ *closest-pair-recurrence* (*length* XS_R)
using *1.IH* **by** *simp*
hence *IHR*: *time* (*closest-pair-rec-tm* XS_R) ≤ *closest-pair-recurrence* (*nat* ⌈*real ?n* / *2*⌉)
using ∗ **by** *simp*

have (*YS*, C_0, C_1) = *val* (*closest-pair-rec-tm ps*)
using *False closest-pair-rec-simps defs* **by** (*auto simp*: *Let-def length-eq-val-length-tm split*!: *prod.split*)
hence *length ps* = *length YS*
using *closest-pair-rec-set-length-sorted-snd closest-pair-rec-eq-val-closest-pair-rec-tm* **by** *auto*
hence *combine-bound*: *time* (*combine-tm* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) (*fst* (*hd* XS_R)) *YS*) ≤ *3* + *18* ∗ *?n*
using *time-combine-tm* **by** *simp*
have *time* (*closest-pair-rec-tm ps*) = *1* + *time* (*length-tm ps*) + *time* (*split-at-tm*

```
(?n div 2) ps) +
          time (closest-pair-rec-tm XS_L) + time (closest-pair-rec-tm XS_R) + time
(merge-tm (λp. snd p) YS_L YS_R) +
           time (combine-tm (C_0L, C_1L) (C_0R, C_1R) (fst (hd XS_R)) YS)
     using time-closest-pair-rec-tm-simps-2[OF False] defs
    by (auto simp del: closest-pair-rec-tm.simps simp add: Let-def split: prod.split)
     also have ... ≤ 7 + 21 * ?n + time (closest-pair-rec-tm XS_L) + time
(closest-pair-rec-tm XS_R)
      using time-merge-tm[of (λp. snd p) YS_L YS_R] L combine-bound by (simp
add: time-length-tm time-split-at-tm)
   also have ... ≤ 7 + 21 * ?n + closest-pair-recurrence (nat ⌊real ?n / 2⌋) +
           closest-pair-recurrence (nat ⌈real ?n / 2⌉)
     using IHL IHR by simp
   also have ... = closest-pair-recurrence (length ps)
     using False by simp
   finally show ?thesis
     by simp
 qed
qed

theorem closest-pair-recurrence:
 closest-pair-recurrence ∈ Θ(λn. n * ln n)
 by (master-theorem) auto

theorem time-closest-pair-rec-bigo:
 (λxs. time (closest-pair-rec-tm xs)) ∈ O[length going-to at-top]((λn. n * ln n) o
length)
proof −
 have 0: ⋀ps. time (closest-pair-rec-tm ps) ≤ (closest-pair-recurrence o length)
ps
  unfolding comp-def using time-closest-pair-rec-conv-closest-pair-recurrence by
auto
 show ?thesis
  using bigo-measure-trans[OF 0] bigthetaD1[OF closest-pair-recurrence] of-nat-0-le-iff
by blast
qed

definition closest-pair-time :: nat ⇒ real where
 closest-pair-time n = 1 + mergesort-recurrence n + closest-pair-recurrence n

lemma time-closest-pair-conv-closest-pair-recurrence:
 time (closest-pair-tm ps) ≤ closest-pair-time (length ps)
 unfolding closest-pair-time-def
proof (induction rule: induct-list012)
 case (3 x y zs)
 let ?ps = x # y # zs
 define xs where xs = val (mergesort-tm fst ?ps)
 have *: length xs = length ?ps
   using xs-def mergesort(3)[of fst ?ps] mergesort-eq-val-mergesort-tm by metis
```

have *time* (*closest-pair-tm ?ps*) = *1* + *time* (*mergesort-tm fst ?ps*) + *time* (*closest-pair-rec-tm xs*)
using *xs-def* **by** (*auto simp del*: *mergesort-tm.simps closest-pair-rec-tm.simps simp add*: *time-simps split*: *prod.split*)
also have ... ≤ *1* + *mergesort-recurrence* (*length ?ps*) + *time* (*closest-pair-rec-tm xs*)
using *time-mergesort-conv-mergesort-recurrence*[*of fst ?ps*] **by** *simp*
also have ... ≤ *1* + *mergesort-recurrence* (*length ?ps*) + *closest-pair-recurrence* (*length ?ps*)
using *time-closest-pair-rec-conv-closest-pair-recurrence*[*of xs*] ∗ **by** *auto*
finally show *?case*
by *blast*
qed (*auto simp*: *time-simps*)

corollary *closest-pair-time*:
closest-pair-time ∈ *O*(λ*n*. *n* ∗ *ln n*)
unfolding *closest-pair-time-def*
using *mergesort-recurrence closest-pair-recurrence sum-in-bigo*(*1*) *const-1-bigo-n-ln-n* **by** *blast*

corollary *time-closest-pair-bigo*:
(λ*ps*. *time* (*closest-pair-tm ps*)) ∈ *O*[*length going-to at-top*]((λ*n*. *n* ∗ *ln n*) *o length*)
proof −
have *0*: ⋀*ps*. *time* (*closest-pair-tm ps*) ≤ (*closest-pair-time o length*) *ps*
unfolding *comp-def* **using** *time-closest-pair-conv-closest-pair-recurrence* **by** *auto*
show *?thesis*
using *bigo-measure-trans*[*OF 0*] *closest-pair-time* **by** *simp*
qed

3.3 Code Export

3.3.1 Combine Step

fun *find-closest-pair-code* :: (*int* ∗ *point* ∗ *point*) ⇒ *point list* ⇒ (*int* ∗ *point* ∗ *point*) **where**
find-closest-pair-code (δ, c_0, c_1) [] = (δ, c_0, c_1)
| *find-closest-pair-code* (δ, c_0, c_1) [*p*] = (δ, c_0, c_1)
| *find-closest-pair-code* (δ, c_0, c_1) (p_0 # *ps*) = (
let (δ′, p_1) = *find-closest-bf-code* p_0 (*take* 7 *ps*) *in*
if δ ≤ δ′ *then*
find-closest-pair-code (δ, c_0, c_1) *ps*
else
find-closest-pair-code (δ′, p_0, p_1) *ps*
)

lemma *find-closest-pair-code-dist-eq*:
assumes δ = *dist-code* c_0 c_1 (Δ, C_0, C_1) = *find-closest-pair-code* (δ, c_0, c_1) *ps*
shows Δ = *dist-code* C_0 C_1

using *assms*
proof (*induction* (δ, c_0, c_1) *ps arbitrary*: δ c_0 c_1 Δ C_0 C_1 *rule*: *find-closest-pair-code.induct*)
case (*3* δ c_0 c_1 p_0 p_2 *ps*)
obtain δ' p_1 **where** δ'*-def*: (δ', p_1) = *find-closest-bf-code* p_0 (*take* 7 (p_2 # *ps*))
by (*metis surj-pair*)
hence *A*: δ' = *dist-code* p_0 p_1
using *find-closest-bf-code-dist-eq*[*of take* 7 (p_2 # *ps*)] **by** *simp*
show *?case*
proof (*cases* $\delta \leq \delta'$)
case *True*
obtain Δ' C_0' C_1' **where** Δ'*-def*: (Δ', C_0', C_1') = *find-closest-pair-code* (δ, c_0, c_1) (p_2 # *ps*)
by (*metis prod-cases4*)
note *defs* = δ'*-def* Δ'*-def*
hence Δ' = *dist-code* C_0' C_1'
using *3.hyps*(*1*)[*of* (δ', p_1) δ' p_1] *3.prems*(*1*) *True* δ'*-def* **by** *blast*
moreover have Δ = Δ' C_0 = C_0' C_1 = C_1'
using *defs True 3.prems*(*2*) **apply** (*auto split*: *prod.splits*) **by** (*metis Pair-inject*)+
ultimately show *?thesis*
by *simp*
next
case *False*
obtain Δ' C_0' C_1' **where** Δ'*-def*: (Δ', C_0', C_1') = *find-closest-pair-code* (δ', p_0, p_1) (p_2 # *ps*)
by (*metis prod-cases4*)
note *defs* = δ'*-def* Δ'*-def*
hence Δ' = *dist-code* C_0' C_1'
using *3.hyps*(*2*)[*of* (δ', p_1) δ' p_1] *A False* δ'*-def* **by** *blast*
moreover have Δ = Δ' C_0 = C_0' C_1 = C_1'
using *defs False 3.prems*(*2*) **apply** (*auto split*: *prod.splits*) **by** (*metis Pair-inject*)+
ultimately show *?thesis*
by *simp*
qed
qed *auto*

declare *find-closest-pair.simps* [*simp add*]

lemma *find-closest-pair-code-eq*:
assumes δ = *dist* c_0 c_1 δ' = *dist-code* c_0 c_1
assumes (C_0, C_1) = *find-closest-pair* (c_0, c_1) *ps*
assumes (Δ', C_0', C_1') = *find-closest-pair-code* (δ', c_0, c_1) *ps*
shows C_0 = C_0' $\wedge$ C_1 = C_1'
using *assms*
proof (*induction* (c_0, c_1) *ps arbitrary*: δ δ' c_0 c_1 C_0 C_1 Δ' C_0' C_1' *rule*: *find-closest-pair.induct*)
case (*3* c_0 c_1 p_0 p_2 *ps*)
obtain p_1 δ_p' p_1' **where** δ_p*-def*: p_1 = *find-closest-bf* p_0 (*take* 7 (p_2 # *ps*))
(δ_p', p_1') = *find-closest-bf-code* p_0 (*take* 7 (p_2 # *ps*))
by (*metis surj-pair*)
hence *A*: δ_p' = *dist-code* p_0 p_1'

```
    using find-closest-bf-code-dist-eq[of take 7 (p2 # ps)] by simp
  have B: p1 = p1'
    using 3.prems(1,2,3) δp-def find-closest-bf-code-eq by auto
  show ?case
  proof (cases δ ≤ dist p0 p1)
    case True
    hence C: δ' ≤ δp'
      by (simp add: 3.prems(1,2) A B dist-eq-dist-code-le)
    obtain C0i C1i Δi' C0i' C1i' where Δi-def:
      (C0i, C1i) = find-closest-pair (c0, c1) (p2 # ps)
      (Δi', C0i', C1i') = find-closest-pair-code (δ', c0, c1) (p2 # ps)
      by (metis prod-cases3)
    note defs = δp-def Δi-def
    have C0i = C0i' ∧ C1i = C1i'
      using 3.hyps(1)[of p1] 3.prems True defs by blast
    moreover have C0 = C0i C1 = C1i
      using defs(1,3) True 3.prems(1,3) apply (auto split: prod.splits) by (metis
Pair-inject)+
    moreover have Δ' = Δi' C0' = C0i' C1' = C1i'
      using defs(2,4) C 3.prems(4) apply (auto split: prod.splits) by (metis
Pair-inject)+
    ultimately show ?thesis
      by simp
  next
    case False
    hence C: ¬ δ' ≤ δp'
      by (simp add: 3.prems(1,2) A B dist-eq-dist-code-le)
    obtain C0i C1i Δi' C0i' C1i' where Δi-def:
      (C0i, C1i) = find-closest-pair (p0, p1) (p2 # ps)
      (Δi', C0i', C1i') = find-closest-pair-code (δp', p0, p1') (p2 # ps)
      by (metis prod-cases3)
    note defs = δp-def Δi-def
    have C0i = C0i' ∧ C1i = C1i'
      using 3.prems 3.hyps(2)[of p1] A B False defs by blast
    moreover have C0 = C0i C1 = C1i
      using defs(1,3) False 3.prems(1,3) apply (auto split: prod.splits) by (metis
Pair-inject)+
    moreover have Δ' = Δi' C0' = C0i' C1' = C1i'
      using defs(2,4) C 3.prems(4) apply (auto split: prod.splits) by (metis
Pair-inject)+
    ultimately show ?thesis
      by simp
  qed
qed auto

fun combine-code :: (int * point * point) ⇒ (int * point * point) ⇒ int ⇒ point
list ⇒ (int * point * point) where
  combine-code (δL, p0L, p1L) (δR, p0R, p1R) l ps = (
    let (δ, c0, c1) = if δL < δR then (δL, p0L, p1L) else (δR, p0R, p1R) in
```

```
    let ps' = filter (λp. (fst p − l)² < δ) ps in
    find-closest-pair-code (δ, c₀, c₁) ps'
  )

lemma combine-code-dist-eq:
  assumes δ_L = dist-code p_0L p_1L δ_R = dist-code p_0R p_1R
  assumes (δ, c₀, c₁) = combine-code (δ_L, p_0L, p_1L) (δ_R, p_0R, p_1R) l ps
  shows δ = dist-code c₀ c₁
  using assms by (auto simp: find-closest-pair-code-dist-eq split: if-splits)

lemma combine-code-eq:
  assumes δ_L' = dist-code p_0L p_1L δ_R' = dist-code p_0R p_1R
  assumes (c₀, c₁) = combine (p_0L, p_1L) (p_0R, p_1R) l ps
  assumes (δ', c₀', c₁') = combine-code (δ_L', p_0L, p_1L) (δ_R', p_0R, p_1R) l ps
  shows c₀ = c₀' ∧ c₁ = c₁'
proof −
  obtain C_0i C_1i Δ_i' C_0i' C_1i' where Δ_i-def:
    (C_0i, C_1i) = (if dist p_0L p_1L < dist p_0R p_1R then (p_0L, p_1L) else (p_0R, p_1R))
    (Δ_i', C_0i', C_1i') = (if δ_L' < δ_R' then (δ_L', p_0L, p_1L) else (δ_R', p_0R, p_1R))
    by metis
  define ps' ps'' where ps'-def:
    ps' = filter (λp. dist p (l, snd p) < dist C_0i C_1i) ps
    ps'' = filter (λp. (fst p − l)² < Δ_i') ps
  obtain C₀ C₁ Δ' C₀' C₁' where Δ-def:
    (C₀, C₁) = find-closest-pair (C_0i, C_1i) ps'
    (Δ', C₀', C₁') = find-closest-pair-code (Δ_i', C_0i', C_1i') ps''
    by (metis prod-cases3)
  note defs = Δ_i-def ps'-def Δ-def
  have ∗: C_0i = C_0i' C_1i = C_1i' Δ_i' = dist-code C_0i' C_1i'
    using Δ_i-def assms(1,2,3,4) dist-eq-dist-code-lt by (auto split: if-splits)
  hence ⋀p. |fst p − l| < dist C_0i C_1i ⟷ (fst p − l)² < Δ_i'
    using dist-eq-dist-code-abs-lt by (metis (mono-tags) of-int-abs)
  hence ps' = ps''
    using ps'-def dist-fst-abs by auto
  hence C₀ = C₀' C₁ = C₁'
    using ∗ find-closest-pair-code-eq Δ-def by blast+
  moreover have C₀ = c₀ C₁ = c₁
   using assms(3) defs(1,3,5) apply (auto simp: combine.simps split: prod.splits)
by (metis Pair-inject)+
  moreover have C₀' = c₀' C₁' = c₁'
   using assms(4) defs(2,4,6) apply (auto split: prod.splits) by (metis prod.inject)+
  ultimately show ?thesis
    by blast
qed
```

3.3.2 Divide and Conquer Algorithm

```
function closest-pair-rec-code :: point list ⇒ (point list ∗ int ∗ point ∗ point)
where
```

```
  closest-pair-rec-code xs = (
    let n = length xs in
    if n ≤ 3 then
      (mergesort snd xs, closest-pair-bf-code xs)
    else
      let (xs_L, xs_R) = split-at (n div 2) xs in
      let l = fst (hd xs_R) in

      let (ys_L, p_L) = closest-pair-rec-code xs_L in
      let (ys_R, p_R) = closest-pair-rec-code xs_R in

      let ys = merge snd ys_L ys_R in
      (ys, combine-code p_L p_R l ys)
  )
  by pat-completeness auto
termination closest-pair-rec-code
  by (relation Wellfounded.measure (λxs. length xs))
     (auto simp: split-at-take-drop-conv Let-def)

lemma closest-pair-rec-code-simps:
  assumes n = length xs ¬ (n ≤ 3)
  shows closest-pair-rec-code xs = (
    let (xs_L, xs_R) = split-at (n div 2) xs in
    let l = fst (hd xs_R) in
    let (ys_L, p_L) = closest-pair-rec-code xs_L in
    let (ys_R, p_R) = closest-pair-rec-code xs_R in
    let ys = merge snd ys_L ys_R in
    (ys, combine-code p_L p_R l ys)
  )
  using assms by (auto simp: Let-def)

declare combine.simps combine-code.simps closest-pair-rec-code.simps [simp del]

lemma closest-pair-rec-code-dist-eq:
  assumes 1 < length xs (ys, δ, c_0, c_1) = closest-pair-rec-code xs
  shows δ = dist-code c_0 c_1
  using assms
proof (induction xs arbitrary: ys δ c_0 c_1 rule: length-induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof (cases ?n ≤ 3)
    case True
    hence (δ, c_0, c_1) = closest-pair-bf-code xs
      using 1.prems(2) closest-pair-rec-code.simps by simp
    thus ?thesis
      using 1.prems(1) closest-pair-bf-code-dist-eq by simp
  next
    case False
```

obtain XS_L XS_R **where** XS_{LR}-*def*: $(XS_L, XS_R) = split\text{-}at\ (?n\ div\ 2)\ xs$
using *prod.collapse* **by** *blast*
define L **where** $L = fst\ (hd\ XS_R)$

obtain YS_L Δ_L C_{0L} C_{1L} **where** YSC_{01L}-*def*: $(YS_L, \Delta_L, C_{0L}, C_{1L}) = closest\text{-}pair\text{-}rec\text{-}code\ XS_L$
using *prod.collapse* **by** *metis*
obtain YS_R Δ_R C_{0R} C_{1R} **where** YSC_{01R}-*def*: $(YS_R, \Delta_R, C_{0R}, C_{1R}) = closest\text{-}pair\text{-}rec\text{-}code\ XS_R$
using *prod.collapse* **by** *metis*

define YS **where** $YS = merge\ (\lambda p.\ snd\ p)\ YS_L\ YS_R$
obtain Δ C_0 C_1 **where** C_{01}-*def*: $(\Delta, C_0, C_1) = combine\text{-}code\ (\Delta_L, C_{0L}, C_{1L})\ (\Delta_R, C_{0R}, C_{1R})\ L\ YS$
using *prod.collapse* **by** *metis*
note *defs* = XS_{LR}-*def L-def* YSC_{01L}-*def* YSC_{01R}-*def YS-def* C_{01}-*def*

have $1 < length\ XS_L$ $length\ XS_L < length\ xs$
using *False 1.prems*(1) *defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHL*: $\Delta_L = dist\text{-}code\ C_{0L}\ C_{1L}$
using *1.IH defs* **by** *metis+*

have $1 < length\ XS_R$ $length\ XS_R < length\ xs$
using *False 1.prems*(1) *defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHR*: $\Delta_R = dist\text{-}code\ C_{0R}\ C_{1R}$
using *1.IH defs* **by** *metis+*

have $*$: $(YS, \Delta, C_0, C_1) = closest\text{-}pair\text{-}rec\text{-}code\ xs$
using *False closest-pair-rec-code-simps defs* **by** (*auto simp*: *Let-def split*: *prod.split*)
moreover have $\Delta = dist\text{-}code\ C_0\ C_1$
using *combine-code-dist-eq IHL IHR* C_{01}-*def* **by** *blast*
ultimately show *?thesis*
using *1.prems*(2) $*$ **by** (*metis Pair-inject*)
qed
qed

lemma *closest-pair-rec-ys-eq*:
assumes $1 < length\ xs$
assumes $(ys, c_0, c_1) = closest\text{-}pair\text{-}rec\ xs$
assumes $(ys', \delta', c_0', c_1') = closest\text{-}pair\text{-}rec\text{-}code\ xs$
shows $ys = ys'$
using *assms*
proof (*induction xs arbitrary*: *ys* c_0 c_1 ys' δ' c_0' c_1' *rule*: *length-induct*)
case (*1 xs*)
let $?n = length\ xs$
show *?case*
proof (*cases* $?n \leq 3$)

case *True*
hence *ys = mergesort snd xs*
using *1.prems(2) closest-pair-rec.simps* **by** *simp*
moreover have *ys' = mergesort snd xs*
using *1.prems(3) closest-pair-rec-code.simps* **by** (*simp add*: *True*)
ultimately show *?thesis*
using *1.prems(1)* **by** *simp*
next
case *False*

obtain XS_L XS_R **where** XS_{LR}*-def*: (XS_L, XS_R) = *split-at* (*?n div 2*) *xs*
using *prod.collapse* **by** *blast*
define *L* **where** *L = fst* (*hd* XS_R)

obtain YS_L C_{0L} C_{1L} YS_L' Δ_L' C_{0L}' C_{1L}' **where** YSC_{01L}*-def*:
(YS_L, C_{0L}, C_{1L}) = *closest-pair-rec* XS_L
(YS_L', Δ_L', C_{0L}', C_{1L}') = *closest-pair-rec-code* XS_L
using *prod.collapse* **by** *metis*
obtain YS_R C_{0R} C_{1R} YS_R' Δ_R' C_{0R}' C_{1R}' **where** YSC_{01R}*-def*:
(YS_R, C_{0R}, C_{1R}) = *closest-pair-rec* XS_R
(YS_R', Δ_R', C_{0R}', C_{1R}') = *closest-pair-rec-code* XS_R
using *prod.collapse* **by** *metis*

define *YS YS'* **where** *YS-def*:
YS = merge (*λp. snd p*) YS_L YS_R
YS' = merge (*λp. snd p*) YS_L' YS_R'
obtain C_0 C_1 Δ' C_0' C_1' **where** C_{01}*-def*:
(C_0, C_1) = *combine* (C_{0L}, C_{1L}) (C_{0R}, C_{1R}) *L YS*
(Δ', C_0', C_1') = *combine-code* (Δ_L', C_{0L}', C_{1L}') (Δ_R', C_{0R}', C_{1R}') *L YS'*
using *prod.collapse* **by** *metis*
note *defs* = XS_{LR}*-def L-def* YSC_{01L}*-def* YSC_{01R}*-def YS-def* C_{01}*-def*

have *1 < length* XS_L *length* XS_L *< length xs*
using *False 1.prems(1) defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHL*: YS_L = YS_L'
using *1.IH defs* **by** *metis*

have *1 < length* XS_R *length* XS_R *< length xs*
using *False 1.prems(1) defs* **by** (*auto simp*: *split-at-take-drop-conv*)
hence *IHR*: YS_R = YS_R'
using *1.IH defs* **by** *metis*

have (*YS*, C_0, C_1) = *closest-pair-rec xs*
using *False closest-pair-rec-simps defs*(*1*,*2*,*3*,*5*,*7*,*9*)
by (*auto simp*: *Let-def split*: *prod.split*)
moreover have (*YS'*, Δ', C_0', C_1') = *closest-pair-rec-code xs*
using *False closest-pair-rec-code-simps defs*(*1*,*2*,*4*,*6*,*8*,*10*)
by (*auto simp*: *Let-def split*: *prod.split*)
moreover have *YS = YS'*

```
    using IHL IHR YS-def by simp
  ultimately show ?thesis
    by (metis 1.prems(2,3) Pair-inject)
 qed
qed

lemma closest-pair-rec-code-eq:
 assumes 1 < length xs
 assumes (ys, c₀, c₁) = closest-pair-rec xs
 assumes (ys', δ', c₀', c₁') = closest-pair-rec-code xs
 shows c₀ = c₀' ∧ c₁ = c₁'
 using assms
proof (induction xs arbitrary: ys c₀ c₁ ys' δ' c₀' c₁' rule: length-induct)
 case (1 xs)
 let ?n = length xs
 show ?case
 proof (cases ?n ≤ 3)
  case True
  hence (c₀, c₁) = closest-pair-bf xs
   using 1.prems(2) closest-pair-rec.simps by simp
  moreover have (δ', c₀', c₁') = closest-pair-bf-code xs
   using 1.prems(3) closest-pair-rec-code.simps by (simp add: True)
  ultimately show ?thesis
   using 1.prems(1) closest-pair-bf-code-eq by simp
 next
  case False

  obtain XS_L XS_R where XS_LR-def: (XS_L, XS_R) = split-at (?n div 2) xs
   using prod.collapse by blast
  define L where L = fst (hd XS_R)

  obtain YS_L C_0L C_1L YS_L' Δ_L' C_0L' C_1L' where YSC_01L-def:
   (YS_L, C_0L, C_1L) = closest-pair-rec XS_L
   (YS_L', Δ_L', C_0L', C_1L') = closest-pair-rec-code XS_L
   using prod.collapse by metis
  obtain YS_R C_0R C_1R YS_R' Δ_R' C_0R' C_1R' where YSC_01R-def:
   (YS_R, C_0R, C_1R) = closest-pair-rec XS_R
   (YS_R', Δ_R', C_0R', C_1R') = closest-pair-rec-code XS_R
   using prod.collapse by metis

  define YS YS' where YS-def:
   YS = merge (λp. snd p) YS_L YS_R
   YS' = merge (λp. snd p) YS_L' YS_R'
  obtain C_0 C_1 Δ' C_0' C_1' where C_01-def:
   (C_0, C_1) = combine (C_0L, C_1L) (C_0R, C_1R) L YS
   (Δ', C_0', C_1') = combine-code (Δ_L', C_0L', C_1L') (Δ_R', C_0R', C_1R') L YS'
   using prod.collapse by metis
  note defs = XS_LR-def L-def YSC_01L-def YSC_01R-def YS-def C_01-def
```

```
    have 1 < length XS_L length XS_L < length xs
      using False 1.prems(1) defs by (auto simp: split-at-take-drop-conv)
    hence IHL: C_0L = C_0L' C_1L = C_1L'
      using 1.IH defs by metis+

    have 1 < length XS_R length XS_R < length xs
      using False 1.prems(1) defs by (auto simp: split-at-take-drop-conv)
    hence IHR: C_0R = C_0R' C_1R = C_1R'
      using 1.IH defs by metis+

    have YS = YS'
     using defs ‹1 < length XS_L› ‹1 < length XS_R› closest-pair-rec-ys-eq by blast
    moreover have Δ_L' = dist-code C_0L' C_1L' Δ_R' = dist-code C_0R' C_1R'
      using defs ‹1 < length XS_L› ‹1 < length XS_R› closest-pair-rec-code-dist-eq
by blast+
    ultimately have C_0 = C_0' C_1 = C_1'
      using combine-code-eq IHL IHR C_01-def by blast+
    moreover have (YS, C_0, C_1) = closest-pair-rec xs
      using False closest-pair-rec-simps defs(1,2,3,5,7,9)
      by (auto simp: Let-def split: prod.split)
    moreover have (YS', Δ', C_0', C_1') = closest-pair-rec-code xs
      using False closest-pair-rec-code-simps defs(1,2,4,6,8,10)
      by (auto simp: Let-def split: prod.split)
    ultimately show ?thesis
      using 1.prems(2,3) by (metis Pair-inject)
  qed
qed

declare closest-pair.simps [simp add]

fun closest-pair-code :: point list ⇒ (point ∗ point) where
  closest-pair-code [] = undefined
| closest-pair-code [-] = undefined
| closest-pair-code ps = (let (-, -, c_0, c_1) = closest-pair-rec-code (mergesort fst ps)
in (c_0, c_1))

lemma closest-pair-code-eq:
  closest-pair ps = closest-pair-code ps
proof (induction ps rule: induct-list012)
  case (3 x y zs)
  obtain ys c_0 c_1 ys' δ' c_0' c_1' where ∗:
    (ys, c_0, c_1) = closest-pair-rec (mergesort fst (x # y # zs))
    (ys', δ', c_0', c_1') = closest-pair-rec-code (mergesort fst (x # y # zs))
    by (metis prod-cases3)
  moreover have 1 < length (mergesort fst (x # y # zs))
    using length-mergesort[of fst x # y # zs] by simp
  ultimately have c_0 = c_0' c_1 = c_1'
    using closest-pair-rec-code-eq by blast+
  thus ?case
```

using $*$ **by** (*auto split*: *prod.splits*)
qed *auto*

export-code *closest-pair-code* **in** *OCaml*
module-name *Verified*

end

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.