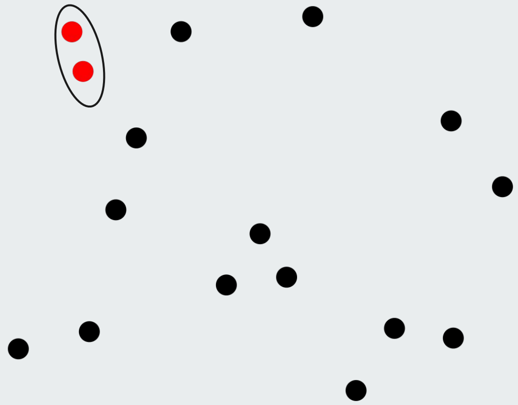
Verification of Closest Pair of Points Algorithm



Aditi Goyal

Suhas Shankar

Yatharth Goswami

EPFL

Problem Definition

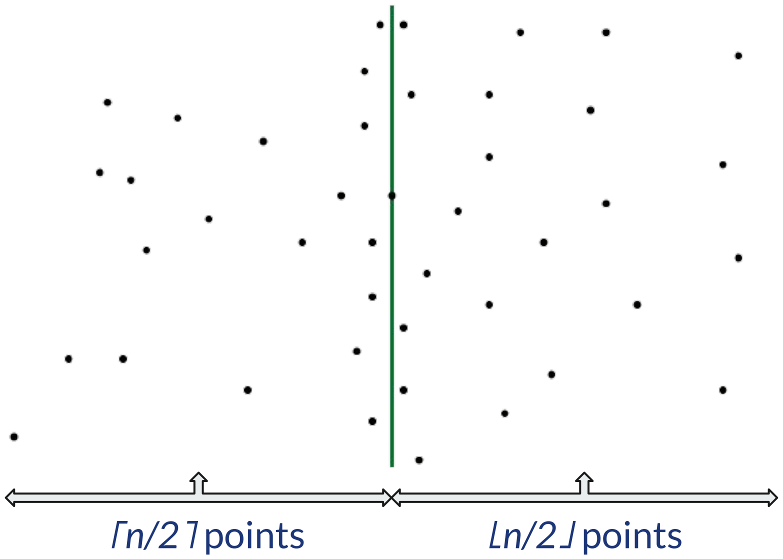
Given a set P of $n > 1$ points in a 2-dimensional plane, compute the pair of points with minimum Euclidean distance amongst them

Algorithms:

- $O(n^2)$ - trivial brute force algorithm
- $O(n \log n)$ - divide and conquer algorithm

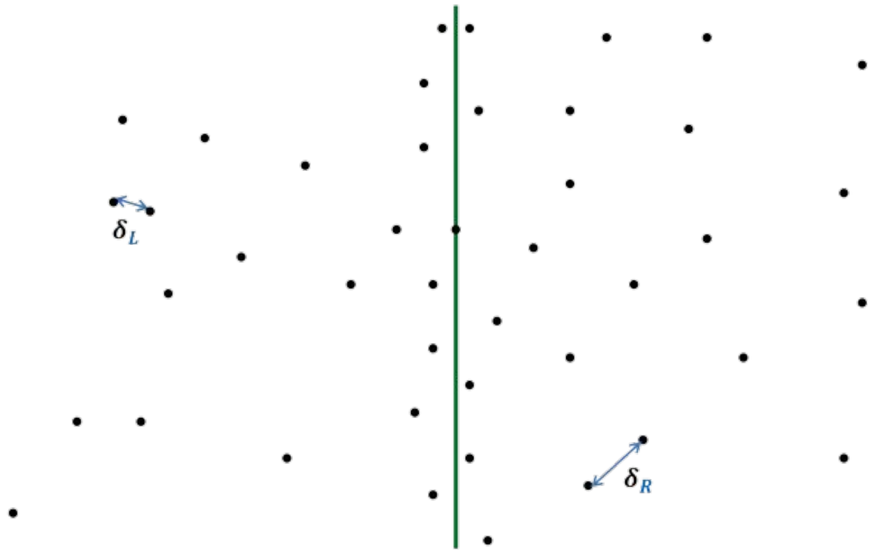
Divide and Conquer Algorithm - Divide Step

- Divide the set of points into two halves using the median x-coordinate



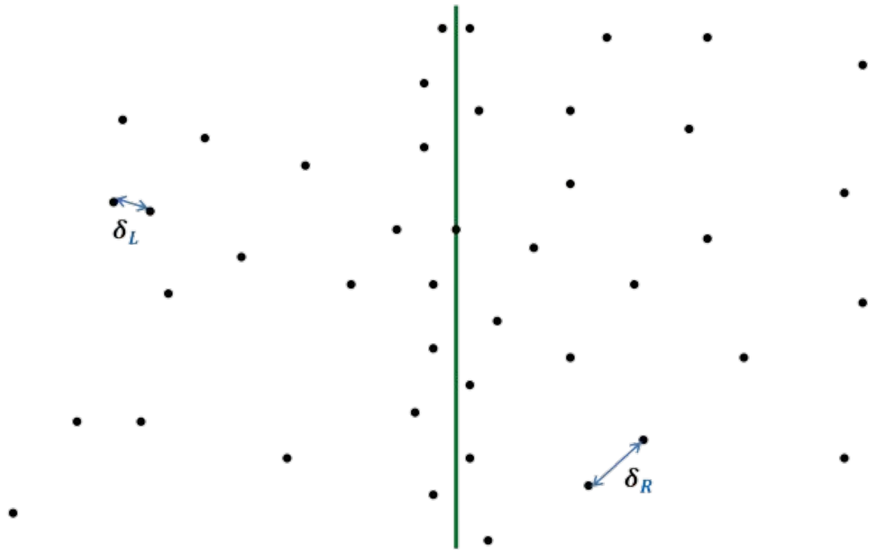
Divide and Conquer Algorithm - Divide Step

- Solve the 2 smaller instances



Divide and Conquer Algorithm - Divide Step

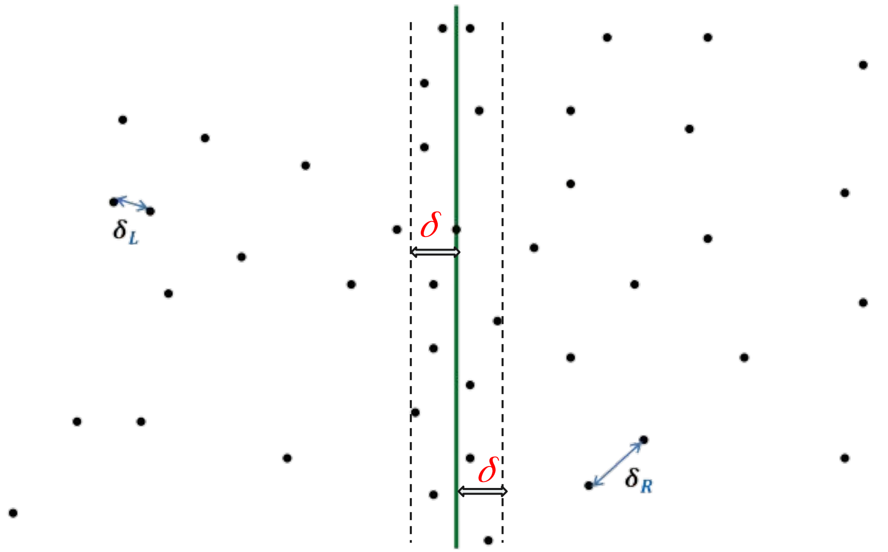
- Solve the 2 smaller instances



$\delta = min(\delta_L, \delta_R)$

Divide and Conquer Algorithm - Combine step

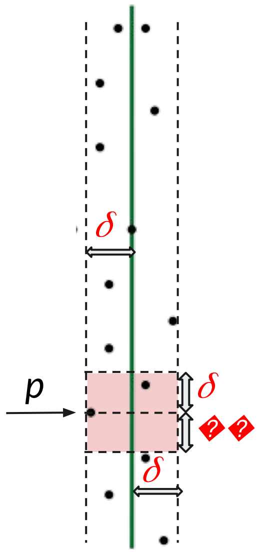
- Only points within the $\boldsymbol{\delta}$ wide strip from the median x-coordinate matter



$\delta = min(\delta_L, \delta_R)$

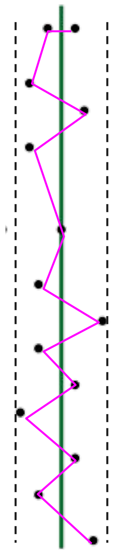
Divide and Conquer Algorithm - Combine step

- For each point, we only need to see points within a 2δ x δ wide rectangle
- Only a constant number of points to check for each point!



$\delta = min(\delta_L, \delta_R)$

Divide and Conquer Algo - Combine step in O(n) time



If the points are sorted by y-coordinates, then combine becomes ***<u>O(n)</u>*** in time, for each point only need to check the next few (**constant**) points!

Onto Verification: Delta Point & Delta Sparsity

- **Delta Point Sparsity** : A set S of points is δ-point sparse with respect to point p_0 iff

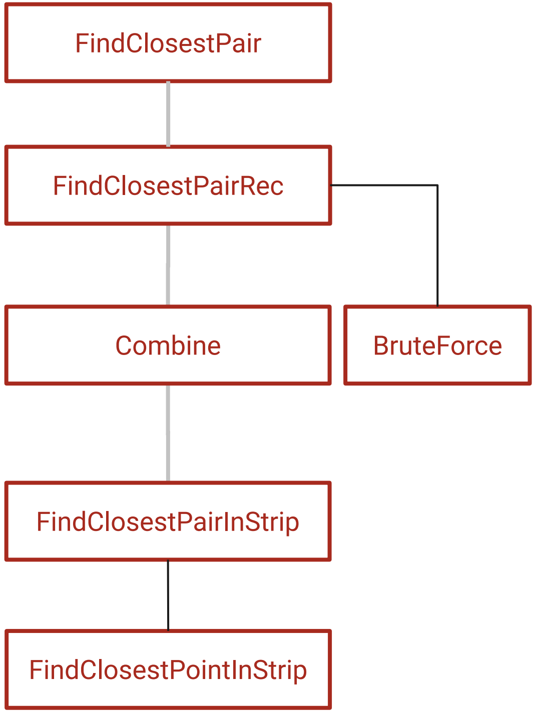
$$p.distance(p_0) \geq \delta \ \forall\, p \in S$$

- **Delta Sparsity** : A set S of points is δ- sparse iff

$$p_0.distance(p_1) \geq \delta \ \forall\, p_0 , p_1 \in S \text{ where } p_0 \neq p_1$$

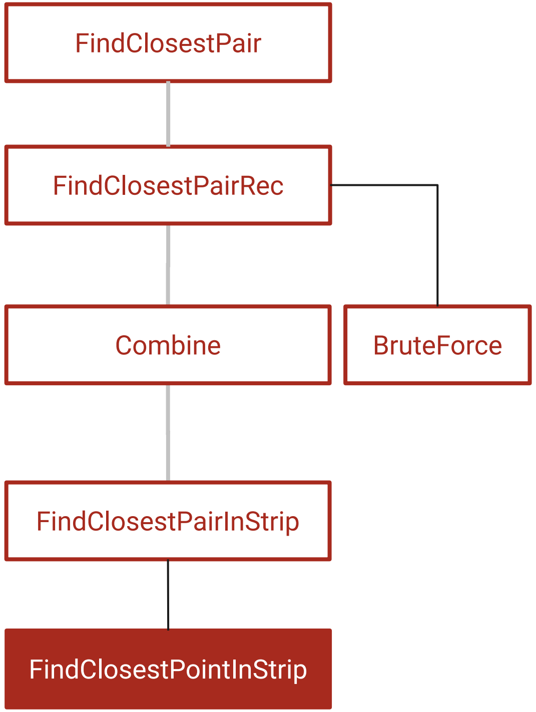
Overview of verified Implementation

- Inspiration from our reference paper implementation (Isabelle)
- Modularise into different functions and prove correctness properties for each
- Will follow bottom up approach for explaining
- Simplified things like distance -> Didn’t take the square root



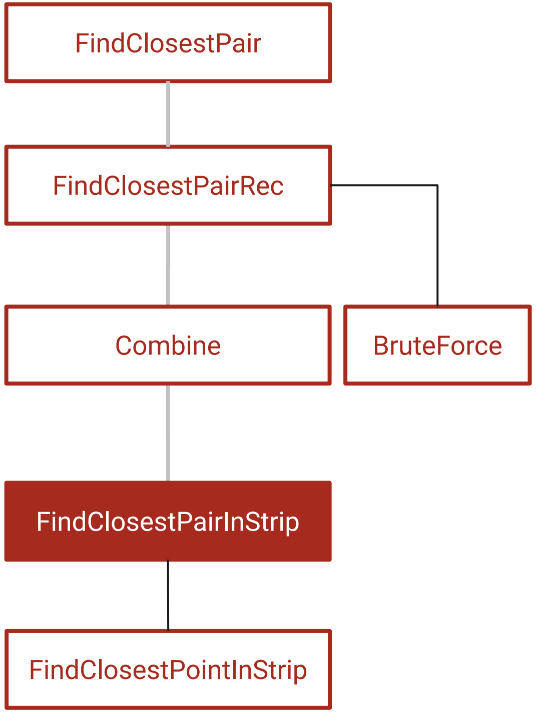
```scala
def findClosestPointInStrip(p: Point)(d: BigInt)(l: List[Point]): Point =
{
    require(!l.isEmpty && isSortedY(l) && p.y <= l.head.y)
    if l.tail.isEmpty then l.head
    else if d <= (l.head.y - p.y)*(l.head.y - p.y) then {
        transitiveDistanceProperty(p, d, l.head, l.tail)
        l.head
    }
    else{
        val p1 = findClosestPointInStrip(p)(min(d, p.distance(l.head)))(l.tail)
        if p.distance(l.head) <= p.distance(p1) then l.head else p1
    }
}
.ensuring(res0 => deltaSparsePoint(min(p.distance(res0), d), p, l) && l.contains(res0))
```

induction



- **Intuitively:** Given a point ***p***, a integer ***d***, and a list ***l***, the function returns
    - the point **closest** to ***p***, in the list ***l*** if any point is within ***d*** distance from ***p***
    - Nearest point according to **y-coordinate** otherwise
- **Requires:** List ***l*** is not empty, it is **sorted** according to y-coordinates
- **Ensures:** List ***l*** is ***min(d, p.distance(res))-point sparse*** with respect to ***p***

```scala
/* Finds the closest pair in strip. If the closest pair has distance atleast
as distance between points in x, then x is returned*/
def findClosestPairInStrip(x: PairPoint)(@induct l: List[Point]):PairPoint =
{
    require(isSortedY(l))
    if l.isEmpty || l.tail.isEmpty then x
    else {
        val p1 = findClosestPointInStrip(l.head)(pairDistance(x))(l.tail)
        assert(deltaSparsePoint(min(p1.distance(l.head), pairDistance(x)), l.head, l.tail))
        if pairDistance(x) <= p1.distance(l.head) then{
            val z = findClosestPairInStrip(x)(l.tail)
            reducingDeltaPreservesPointSparsity(pairDistance(x), pairDistance(z), l.head, l.tail)
            z
        }
        else {
            val z = findClosestPairInStrip((l.head, p1))(l.tail)
            reducingDeltaPreservesPointSparsity(l.head.distance(p1), pairDistance(z), l.head, l.
            tail)
            z
        }
    }
}
.ensuring(res0 => deltaSparse(pairDistance(res0), l) && pairDistance(res0) <= pairDistance(x) &&
(res0 == x || (l.contains(res0._1) && l.contains(res0._2))))
```

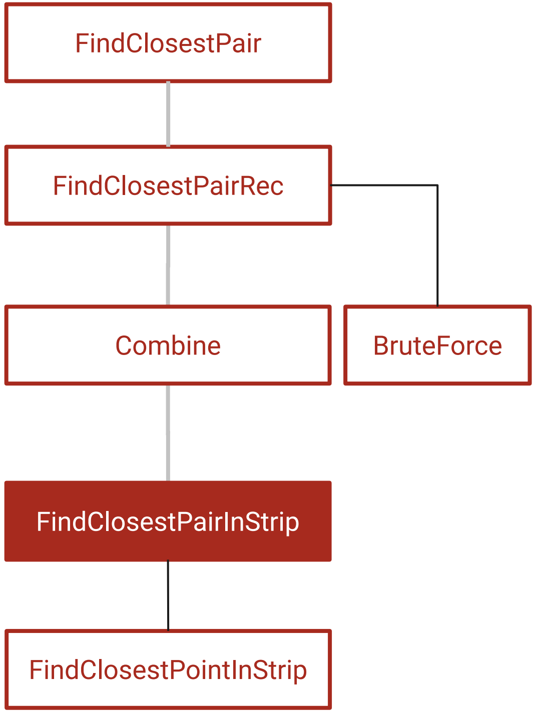


- **Intuition:** Given a list of points *l*, and a pair of points **x**, returns ***x*** or <u>closest pair of points</u> in *l*, depending on which is closer.

- **Requires:** *l* is <u>sorted</u> according to y-coordinates

```scala
/* Finds the closest pair in strip. If the closest pair has distance atleast
as distance between points in x, then x is returned*/
def findClosestPairInStrip(x: PairPoint)(@induct l: List[Point]):PairPoint =
{
    require(isSortedY(l))
    if l.isEmpty || l.tail.isEmpty then x
    else {
        val p1 = findClosestPointInStrip(l.head)(pairDistance(x))(l.tail)
        assert(deltaSparsePoint(min(p1.distance(l.head), pairDistance(x)), l.head, l.tail))
        if pairDistance(x) <= p1.distance(l.head) then{
            val z = findClosestPairInStrip(x)(l.tail)
            reducingDeltaPreservesPointSparsity(pairDistance(x), pairDistance(z), l.head, l.tail)
            z
        }
        else {
            val z = findClosestPairInStrip((l.head, p1))(l.tail)
            reducingDeltaPreservesPointSparsity(l.head.distance(p1), pairDistance(z), l.head, l.
            tail)
            z
        }
    }
}
.ensuring(res0 => deltaSparse(pairDistance(res0), l) && pairDistance(res0) <= pairDistance(x) &&
(res0 == x || (l.contains(res0._1) && l.contains(res0._2))))
```

- **Ensures:**
  - *l* is <u>distance between returned pair of points - sparse</u>
  - Distance between the pair of points returned is <u>at most</u> the distance between points in *x*
  - Returned pair of points is either *x* or is contained in *l*

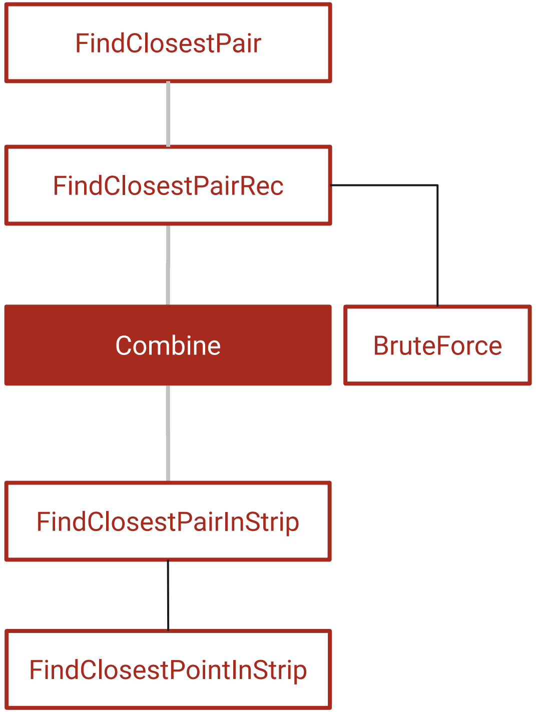


```
/* Combining answers from left and right halves separated by x-coordinate
div */
def combine(lpoint: PairPoint)(rpoint: PairPoint)(div: BigInt)(l: List
[Point]): PairPoint = {
    require(isSortedY(l) && l.contains(lpoint._1) && l.contains(lpoint.
    _2) && l.contains(rpoint._1) && l.contains(rpoint._2))
    val z = compare(lpoint, rpoint)
    val d = pairDistance(z)
    val l2 = l.filter(p => p.distance(Point(div, p.y)) < d)
    ghost { filterSorted(l, p => p.distance(Point(div, p.y)) < d) }
    findClosestPairInStrip(z)(l2)
}.ensuring(res0 => deltaSparse(pairDistance(res0), l.filter(p => p.distance
(Point(div, p.y)) < pairDistance(compare(lpoint, rpoint)))) && pairDistance
(res0) <= pairDistance(compare(lpoint, rpoint)) && l.contains(res0._1) && l.
contains(res0._2))
```

- **Intuition:** Given a list of points ***l***, the dividing x-coordinate ***div***, and closest pair of points on the left half and right half
    - Returns the closest pair of points in the list ***l***

- **Requires:** ***l*** is sorted according to y-coordinates and points in ***lpoint*** and ***rpoint*** are in ***l***

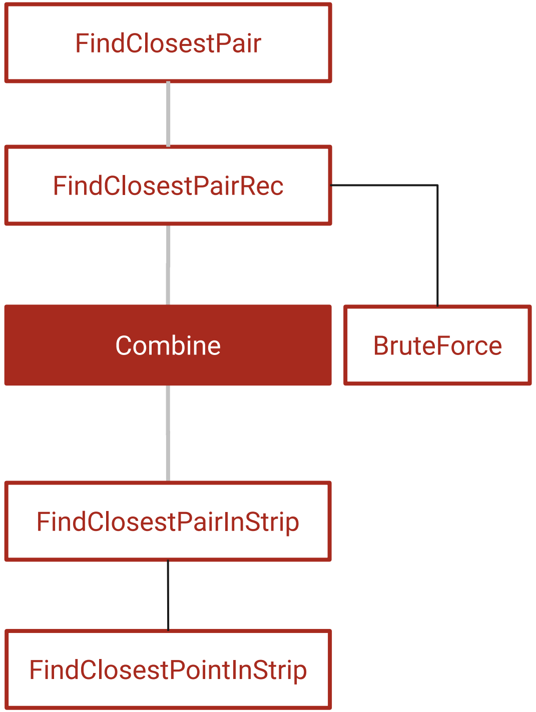
- **Ensures:** ***l*** is distance between the returned pair of points-sparse and the points are either contained in ***l***, or is either ***lpoint*** or ***rpoint***



Important Lemmas - divideAndConquer Lemma

Lemma 3.2 $ps.contains(p_0) \wedge ps.contains(p_1) \wedge p_0 \neq p_1 \wedge d(p_0, p_1) < \delta \wedge$
$(\forall p \in P_L \,.\, p.\_1 \leq l) \wedge sparse(\delta, P_L) \wedge$
$(\forall p \in P_R \,.\, p.\_1 > l) \wedge sparse(\delta, P_R) \wedge$
$ps.content = (P_L + +P_R).content \wedge ps' = ps.filter((p) => d(p, (l, p.\_2)) < \delta)$
$\implies ps'.contains(p_0) \wedge ps'.contains(p_1)$

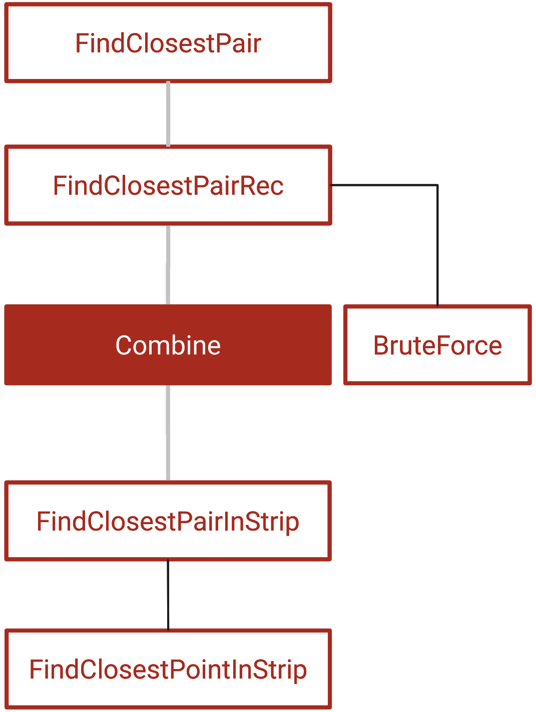
- Assuming $p_0 \in P_L$, conclude $p_1 \in P_R$
- If $p_0 \notin ps'$, distance of all points in P_R from p_0 is at least δ!
- Proceed similarly for other cases
- **Conclusion:** looking for closest pair in strip suffices!



Important Lemmas - combineLemma

Lemma 3.3 $sorted\_y(ps) \land ps.content = (P_L ++ P_R).content \land$
$(\forall p\,.\, P_L.contains(p) \implies p.\_1 \leq l) \land sparse(d(p_{0L}, p_{1L}), P_L) \land$
$(\forall p\,.\, P_R.contains(p) \implies p.\_1 > l) \land sparse(d(p_{0R}, p_{1R}), P_R) \land$
$(p_0, p_1) = combine((p_{0L}, p_{1L}), (p_{0R}, p_{1R}), l, ps)$
$\implies sparse((d(p_0, p_1), ps))$

- Assume that the sparsity condition is **false**
- Use Lemma 3.2 : The divide and conquer lemma
- The strip was **sparse** by **ensuring** clause of **combine!**



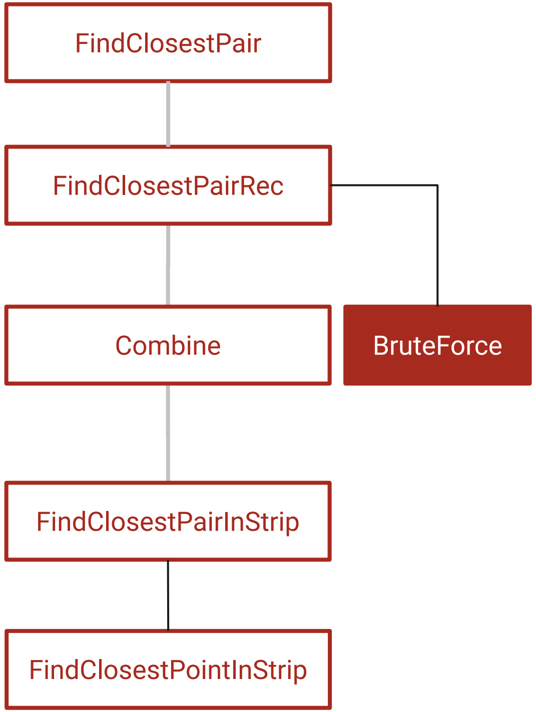
```scala
/* Finds the point closest to p in list l (sorted by y-coordinates)
If there is no point which has distance less than d, then first point
having difference in y-coordinate from p of atleast d is returned */
 def bruteForce(l: List[Point]): (List[Point], PairPoint) =
 {
    require(l.size <= 3 && l.size >= 2)
    val z = mergeSortY(l)
    if l.size == 2 then (z, (l(0), l(1)))
    else {
        val a = l(0).distance(l(1))
        val b = l(0).distance(l(2))
        val c = l(1).distance(l(2))

        /* Explicitly make conditions for verification process*/
        if(a <= b  && b <= c){
            (z, (l(0), l(1)))
        }
        else if(a <= c && c <= b){
            (z, (l(1), l(0)))
        }
        else if(b <= a && a <= c){
            (z, (l(0), l(2)))
        }
        else if(b <=c && c <= a){
            (z, (l(2), l(0)))
        }
        else if(c <= a && a <= b){
            (z, (l(1), l(2)))
        }
        else{
            assert(c <= b && b <= a)
            (z, (l(2), l(1)))
        }
    }
 }
```

- **Intuition:** Given a list *l*, returns a tuple containing the list of points **sorted** according to y-coordinates and **the closest pair of points** in *l*.

- **Requires:** Size of list is at most **3** and at least **2**

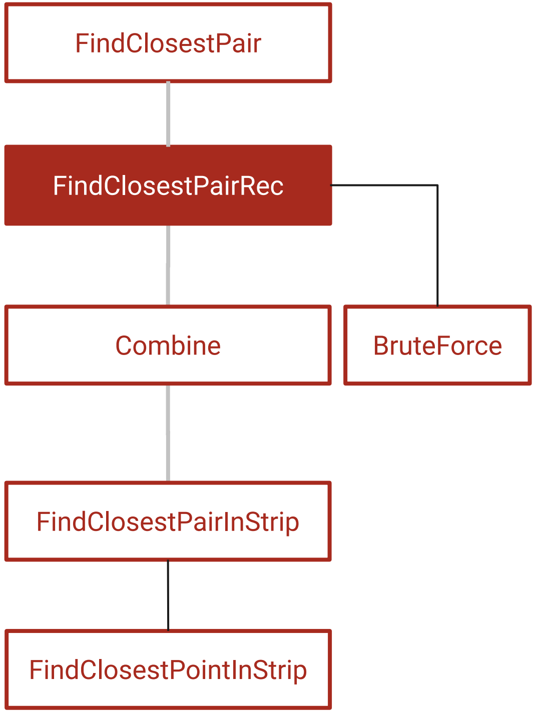
- **Ensures:** The list returned is **sorted** with respect to y-coordinates, and the original list is **distance between the returned pair of points- sparse**. Also, that the returned pair of points lies in *l*



```scala
/* Find closest pair of points in list l sorted by x-coordinates.
Also returns l sorted by y-coordinates */
def findClosestPairRec(l: List[Point]): (List[Point], PairPoint) = {
    require(l.size >= 2 && isSortedX(l))
    decreases(l.size)
    if l.size <= 3 then bruteForce(l)
    else{
        val (left_half, right_half) = l.splitAtIndex(l.size/2)
        split(l, l.size/2)
        val (lsorted, lpoint) = findClosestPairRec(left_half)
        val (rsorted, rpoint) = findClosestPairRec(right_half)
        val sortedList = mergeY(lsorted, rsorted)
        val res = combine(lpoint)(rpoint)(right_half.head.x)(sortedList)
        combineLemma(sortedList, left_half, right_half, right_half.head.x,
        lpoint, rpoint, res)
        subsetPreservesDeltaSparsity(pairDistance(res), sortedList, l)
        (sortedList, res)
    }
}.ensuring(res0 => res0._1.content == l.content && isSortedY(res0._1) &&
deltaSparse(pairDistance(res0._2), l) && l.contains(res0._2._1) && l.contains
(res0._2._2))
```

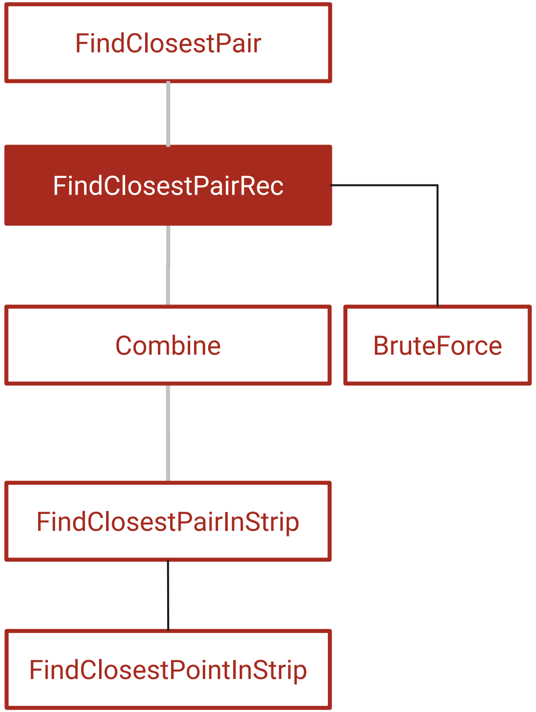
- **Intuition:** Given a list *l* of points sorted, according to x-coordinates, it returns a tuple containing the same list sorted according to y-coordinates and the closest pair of points in *l*

- **Requires:** List *l* contains at least 2 points and it is sorted according to x-coordinate.



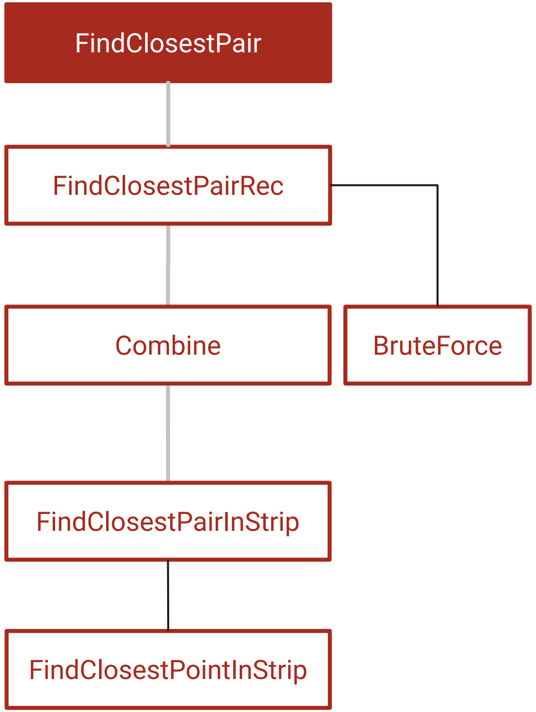
```scala
/* Find closest pair of points in list l sorted by x-coordinates.
Also returns l sorted by y-coordinates */
def findClosestPairRec(l: List[Point]): (List[Point], PairPoint) = {
    require(l.size >= 2 && isSortedX(l))
    decreases(l.size)
    if l.size <= 3 then bruteForce(l)
    else{
        val (left_half, right_half) = l.splitAtIndex(l.size/2)
        split(l, l.size/2)
        val (lsorted, lpoint) = findClosestPairRec(left_half)
        val (rsorted, rpoint) = findClosestPairRec(right_half)
        val sortedList = mergeY(lsorted, rsorted)
        val res = combine(lpoint)(rpoint)(right_half.head.x)(sortedList)
        combineLemma(sortedList, left_half, right_half, right_half.head.x,
        lpoint, rpoint, res)
        subsetPreservesDeltaSparsity(pairDistance(res), sortedList, l)
        (sortedList, res)
    }
}.ensuring(res0 => res0._1.content == l.content && isSortedY(res0._1) &&
deltaSparse(pairDistance(res0._2), l) && l.contains(res0._2._1) && l.contains
(res0._2._2))
```

- **Ensures:**
    - The list returned contains **the same elements** as those of original list
    - It is **sorted** according to y-coordinates
    - List **contains** the pair of points returned and,
    - List is **distance between the returned pair of points-sparse**



```scala
/* Find closest pair of points in list l */
def findClosestPair(l: List[Point]): PairPoint = {
    require(l.size >= 2)
    val p = findClosestPairRec(mergeSortX(l))._2
    subsetPreservesDeltaSparsity(pairDistance(p), mergeSortX
    (l), l)
    p
}.ensuring(res0 => deltaSparse(pairDistance(res0), l) && l.
contains(res0._1) && l.contains(res0._2))
```

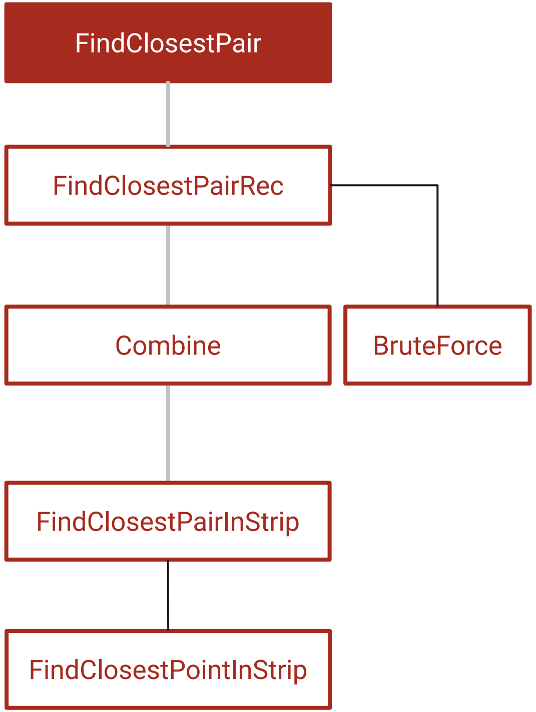
- **Intuition:** The main function to be used to find the closest pair of points in a list *l*
- **Requires:** List *l* contains at least **2** points
- **Ensures:**
  - *l* is distance between returned pair of points-sparse
  - Returned points are contained in *l*



Last piece in the puzzle

- Still left to prove that **if all elements in the list are distinct**, then **distinct points** will be returned by **FindClosestPair**

- Wrote a host of other **lemmas** corresponding to every function **returning distinct pair of points**, if given **list of distinct points**



Last piece in the puzzle

Final specification for FindClosestPair:

1. **Delta sparsity**
2. Resulting points **in the list**
3. If **list is distinct**, resulting points are **distinct**

```scala
@ghostAnnot
def corollary1(xs: List[Point], p: PairPoint) = {
    require(1 < xs.length && p == findClosestPair(xs))
}.ensuring(_ => deltaSparse(pairDistance(p), xs))

@ghostAnnot
def theorem2(xs: List[Point], p0: Point, p1: Point) = {
    require(1 < xs.length && (p0, p1) == findClosestPair(xs))
}.ensuring(_ => xs.contains(p0) && xs.contains(p1))

@ghostAnnot
def theorem3(xs: List[Point], p0: Point, p1: Point) = {
    require(1 < xs.length && isDistinct(xs) && (p0, p1) == findClosestPair(xs))
    mergeSortXDistinctLemma(xs)
    val l = mergeSortX(xs)
    val res = findClosestPairRec(l)
    findClosestPairRecDistinctLemma(l, res._1, res._2)
}.ensuring(_ => p0!=p1)
```

Verification summary

Total project around **1.5k** lines of code and takes around **2-4** minutes to verify without cache

```
Info  ] Utils.scala:54:17:   wholeImpliesSubsetLemma   precond. (call instantiateForAll[T](l1, predicate, h...)  valid  nativez3  0.0
Info  ] Utils.scala:54:50:   wholeImpliesSubsetLemma   precond. (call head[T](l3))                                valid  nativez3  0.0
Info  ] Utils.scala:55:17:   wholeImpliesSubsetLemma   postcondition                                              valid  nativez3  0.0
Info  ] Utils.scala:55:24:   wholeImpliesSubsetLemma   body assertion                                             valid  nativez3  0.0
Info  ] Utils.scala:55:34:   wholeImpliesSubsetLemma   precond. (call head[T](l3))                                valid  nativez3  0.0
Info  ] Utils.scala:56:17:   wholeImpliesSubsetLemma   measure decreases                                          valid  nativez3  0.0
Info  ] Utils.scala:56:17:   wholeImpliesSubsetLemma   precond. (call wholeImpliesSubsetLemma[T](l1, l2, ta...)   valid  nativez3  0.0
Info  ] Utils.scala:56:49:   wholeImpliesSubsetLemma   precond. (call tail[T](l3))                                valid  nativez3  0.0
Info  ] Utils.scala:58:24:   wholeImpliesSubsetLemma   body assertion                                             valid  nativez3  0.0
Info  ] Utils.scala:58:44:   wholeImpliesSubsetLemma   precond. (call head[T](l3))                                valid  nativez3  0.0
Info  ] Utils.scala:59:17:   wholeImpliesSubsetLemma   precond. (call instantiateForAll[T](l2, predicate, h...)  valid  nativez3  0.0
Info  ] Utils.scala:59:50:   wholeImpliesSubsetLemma   precond. (call head[T](l3))                                valid  nativez3  0.0
Info  ] Utils.scala:60:17:   wholeImpliesSubsetLemma   postcondition                                              valid  nativez3  0.0
Info  ] Utils.scala:60:24:   wholeImpliesSubsetLemma   body assertion                                             valid  nativez3  0.0
Info  ] Utils.scala:60:34:   wholeImpliesSubsetLemma   precond. (call head[T](l3))                                valid  nativez3  0.0
Info  ] Utils.scala:61:17:   wholeImpliesSubsetLemma   measure decreases                                          valid  nativez3  0.0
Info  ] Utils.scala:61:17:   wholeImpliesSubsetLemma   precond. (call wholeImpliesSubsetLemma[T](l1, l2, ta...)   valid  nativez3  0.0
Info  ] Utils.scala:61:49:   wholeImpliesSubsetLemma   precond. (call tail[T](l3))                                valid  nativez3  0.0
Info  ] Utils.scala:63:10:   wholeImpliesSubsetLemma   postcondition                                              valid  nativez3  0.0
Info  ]
Info  ] total: 1475 valid: 1475 (0 from cache, 8 trivial) invalid: 0    unknown: 0    time:  152.29
Info  ]
Info  ] Verification pipeline summary:
Info  ]   anti-aliasing, imperative elimination, nativez3
Info  ] Shutting down executor service.
```

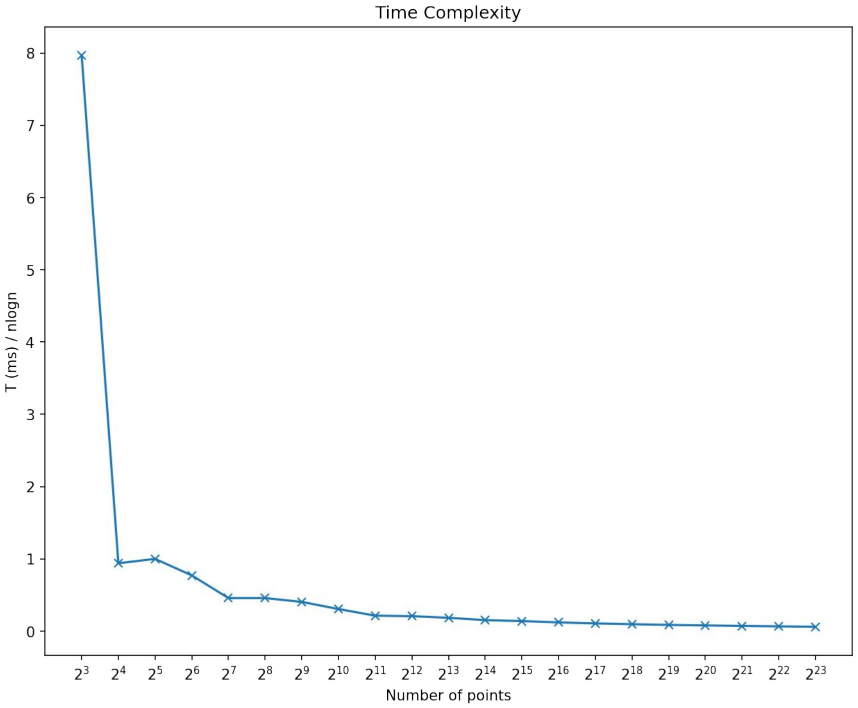
Time complexity

Large n ⇒ Stack overflow

Made our implementation Tail recursive

Stainless list operations not Tail recursive, however

Ran it using Scala lists.



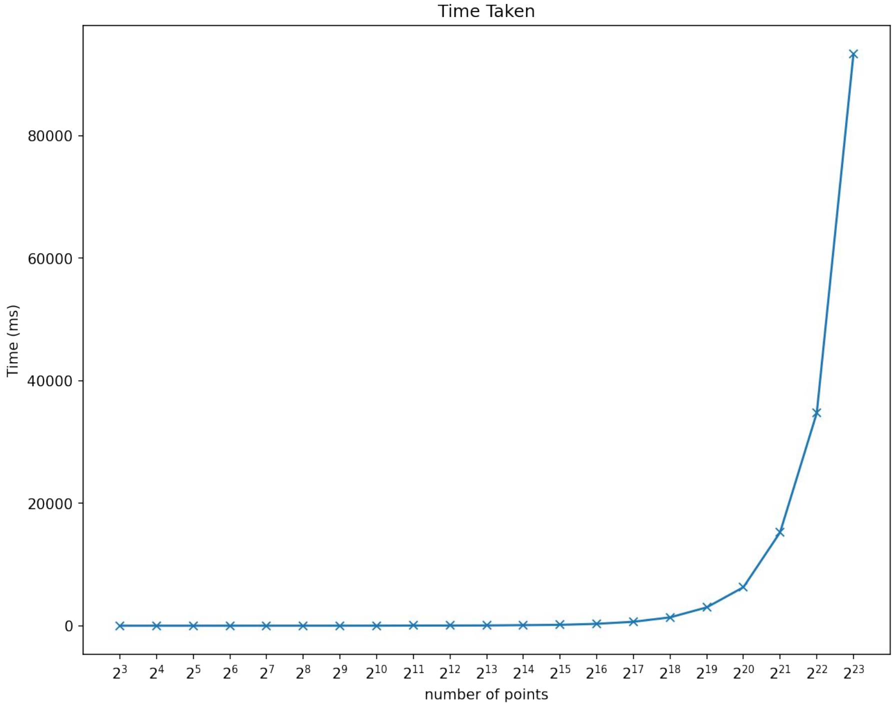
Time complexity

Large n ⇒ Stack overflow

Made our implementation Tail recursive

Stainless list operations not Tail recursive, however

Ran it using Scala lists.



Conclusion and future works

- Verified a **fully functional** implementation of the closest pair of points in a 2D-plane, **divide and conquer** algorithm

- Formally verifying the **time complexity** is a possible future work

- Another possible future work - Verifying algorithm for points in **higher** dimensions

- Possibly making the stainless **list methods tail recursive** for benchmarking

Acknowledgement

- Faced issues due to some bugs related to first order logic in Stainless
- Special thanks to **Mario Bucev** for helping us with the bug and suggesting its alternatives

References

- Thomas H. Cormen et al. Introduction to Algorithms, Third Edition. 3rd. The MIT Press, 2009. isbn: 0262033844.
- Martin Rau and Tobias Nipkow. “Verification of Closest Pair of Points Algorithms”. In: Automated Reasoning. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, 2020, pp. 341–357. isbn: 978-3-030-51054-1.
- Stainless documentation: *https://epfl-lara.github.io/stainless/*

Thank you!

Questions?