1 Preliminaries

We'll give two algorithms for the following *closest pair* problem:

> Given n points in the plane, find the pair of points that is the closest together.

The first algorithm is a deterministic divide and conquer and runs in $O(n \log n)$. The second one is random incremental and runs in expected time $O(n)$. In this lecture we make the following assumptions:

> We assume the points are presented as real number pairs (x, y).
> We assume arithmetic on reals is accurate and runs in $O(1)$ time.
> We will assume that we can take the floor function of a real.
> We also assume that hashing is $O(1)$ time.

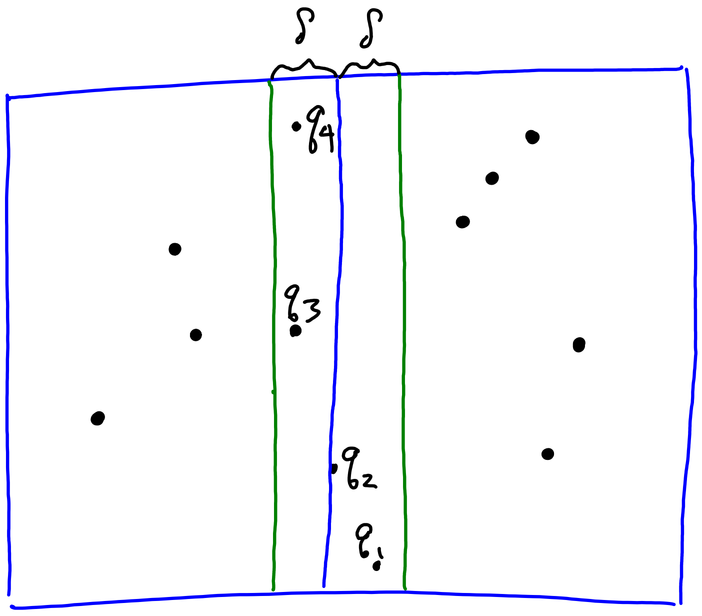
These assumptions (in this context) are resonable, because the algorithms will not abuse this power.

2 $O(n \log n)$ Divide and Conquer Algorithm

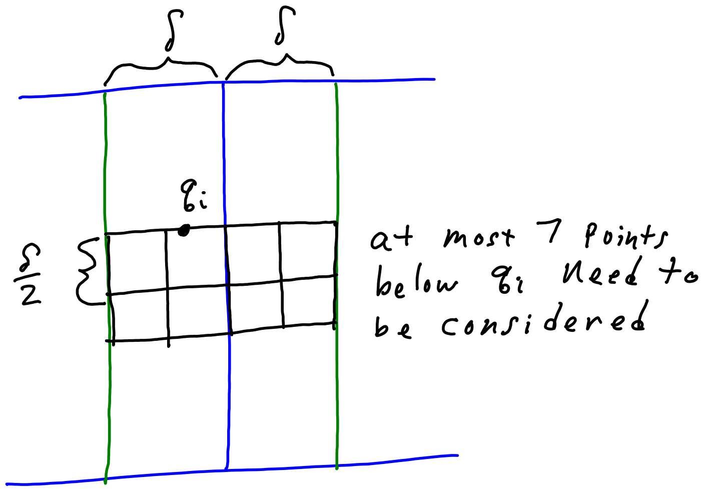
First we present the algorithm. Then we prove that it works. Then we analyze its running time.

```
ClosestPair $(p_1, p_2, \ldots, p_n)$:
    // The points are in sorted order by $x$.
    // We also have the points in a different list ordered by $y$

    if $n \leq 3$ then solve and return the answer.
    let $m = \lfloor n/2 \rfloor$
    let $\delta = \min(\text{ClosestPair}(p_1, \ldots, p_m), \text{ClosestPair}(p_{m+1}, \ldots, p_n))$

    form a list $q$ of the points (sorted by increasing y) that are
    within $\delta$ of the $x$ coordinate of $p_m$. Call these points $q_1, q_2 \ldots q_k$

    Now we compute $d_i$, the minimum distance between $q_i$ and
    all the $q$s below it. We do this for all $1 \leq i \leq k$
    $d_i$ = min distance from $q_i$ to $q_{i-1}, q_{i-2}, \ldots, q_j$
        where we stop when $j$ gets to 1, or the $y$ coordinate gets
        too small: $q_j.y < q_i.y - \delta$

    Return $\min(d_1, \ldots, d_k, \delta)$
```

The divide and conquer approach is obvious. The closest pair is either within the left half, within the right half, or it has one endpoint in the left half and one in the right half.



We need to determine if there is a pair that straddles the dividing line that has a distance less than δ. This is accomplished by the inner loop. It's obviously correct, because any pairs that are not considered are too far apart. (Their y coordinates differ by at least δ.) The only tricky part is proving that the inner loop (trying $q_{i-1}, \ldots, q_j$) finds a stopping value of j in $O(1)$ time.



In this figure there is a four (across) by two (down) grid of squares of size $\delta/2$. The top of the grid goes through q_i. Note that each of these squares can contain at most one of the points (interior or on its boundary), because any two points in the square are at most $\delta/\sqrt{2} < \delta$ apart. Therefore any q that is below the bottom of this grid is at least δ away from q_i, and thus has no effect on the closest pair distance. This proves that there are at most 7 points besides below q_i that are tested before the loop terminates. So for each i this search is $O(1)$ time.

The initiation phase sorts by x and also by y. This is $O(n \log n)$. Subsequent phases just filter these sorted lists, and no further sorting is done.

So the algorithm partitions the points ($O(n)$), does two recursive calls of $n/2$ in size, scans the points again to form the list q ($O(n)$), then scans the list q looking for the closest side-crossing pair ($O(n)$).

So we get the classic divide and conquer recurrence:

$$T(n) = 2T(n/2) + n$$

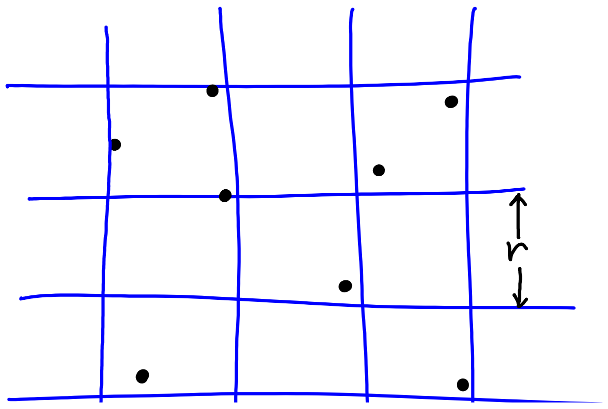
which solves to $O(n \log n)$.

3 Sariel Har-Peled's Randomized $O(n)$ Algorithm for closest pair

For any set of points P, let $\mathrm{CP}(P)$ be the closest pair distance in P.

We're going to define a "grid" data structure, and an API for it. The grid (denoted G) stores a set of points, (we'll call P) and also stores the closest pair distance r for those points. The number r is called "the grid size of G". Here's the API:

MakeGrid(p, q):	Make and return a new grid containing points p and q using $r = \lvert p - q \rvert$ as the initial grid size.
Lookup(G, p):	p is a point, G is a grid. This returns two types of answers. Let r' be the closest distance from p to a point in P. If $r' < r$ then return r'. If $r' \geq r$ return "Not Closest". Note that Lookup() does not need to compute r' if $r' \geq r$.
Insert(G, p):	G is a grid. p is a new point not in the grid. This inserts p into the grid. It returns the current grid size.



The figure above shows a portion of the grid. Suppose a new point p lands in the middle square of this grid. Now to determine if it is closer to another point than the grid size, all we have to do is examine the points in each of the nine boxes shown. (In this case it is nine points.) Since $\mathrm{CP}(P)$ is the grid size, this means that each box has at most four points in it, and we only have to examine a total of at most 36 points.

Here's how we implement these. First define a function Boxify$((x, y), r)$. This returns the integer point $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$.

The data structure maintains a hash table whose keys are integer pairs. The values in the hash table are lists of points from P. So the key (i, j) (also called a *box*) in the hash table stores all the points of P whose Boxify() value is (i, j). Also, it is inductively maintained that the grid size r is always equal to CP(P), the closest pair distance for the set of points being stored.

MakeGrid(p, q) is trivial. Just insert p and q into a new table with $r = |p - q|$.

Lookup(G, p) computes Boxify(p, r). It then looks in that box, and the 8 surrounding ones, and computes the distance between p and the closest one of these. Call this number r'. If $r' < r$, then we return r'. If $r' > r$ then return "Not Closest". This works because we know that if there is a point closer to p than r, it must be in one of the 9 boxes that are searched by this function. Also note that the running time of this is O(1) because it does 9 lookups in the hash table, and the total number of points it has to consider is at most 36. (A box contains at most 4 points.)

Insert(G, p) works as follows. It first does a Lookup(G, p). If the result is "Not Closest" it just inserts p into the data structure into box Boxify(p, r). This is $O(1)$ time. On the other hand if the Lookup() returns $r' < r$, then the algorithm rehashes every point into a new hash table based on the grid size being r'. This takes $O(i)$ time if there are i points now being stored in the data structure.

These algorithms are clearly correct, simply by virtue of the fact that at any point in time the grid size r is equal to the CP(P) where P is the set of points in the data structure. This is preserved by all the operations.

We can now complete the description of the algorithm.

```
Randomized-CP($P$):
    Randomly permute the points. Call the new ordering $p_1, p_2, \ldots, p_n$.
    $G$ = Makegrid($p_1, p_2$)
    for $i = 3$ to $n$ do
        $r$ = Insert($G, p_i$)
    done
    return $r$
```

Claim: This algorithm computes CP(P).

Proof: Follows from the definition of the API. ■

Claim: The algorithm runs in expected $O(n)$ time.

Proof: Recall the time to do Insert() is $O(1)$ if the grid size does not change, and $O(i)$ (i = the number of points in the grid) if the grid size does change.

Consider running the algorithm backwards. Here we are deleting points in order $p_n, p_{n-1}, \ldots, p_3$. When deleting point i, the operation is $O(1)$ if the closest pair distance does not change, and $O(i)$ if it does. In general if you remove a random point from a set of i points, the probability that the closest pair distance changes is at most $2/i$. (Because if there is just one pair with that closest distance, then deleting one of them is the only way to increase the closest pair distance. In other configurations, the probability is lower.)

So the removal is costly (i.e. $O(i)$) with probability at most $2/i$, and cheap $O(1)$ with the remaining probability. Therefore the expected cost of a step is $O(1)$. Thus the expected cost of the entire algorithm is $O(n)$. ■

Below is code implementing this algorithm in Ocaml.

```
(*  Sariel Har-Peled's linear time algorithm for closest pairs.
    Danny Sleator, Nov 2014
*)

let sq x = x *. x

(* The function below takes an array of points (float*float), and returns
   the distance between the closest pair of points.  *)
let closest_pair p =
  let n = Array.length p in

  let dist i j =
    let (xi,yi) = p.(i) in
    let (xj,yj) = p.(j) in
    sqrt ((sq (xi -. xj)) +. (sq (yi -. yj)))
  in

  let truncate x r = int_of_float (floor (x /. r)) in
  let boxify (x,y) r = (truncate x r, truncate y r) in

  let getbox h box = try Hashtbl.find h box with Not_found -> [] in

  let add_to_h h i box =
    Hashtbl.replace h box (i::(getbox h box))
  in

  let make_grid i r =
    (* put points p.(0) ... p.(i) into a new grid of size r.
       it has already been established that the closest pair
       in that point set has distance r *)
    let h = Hashtbl.create 10 in

    for j=0 to i do
      add_to_h h j (boxify p.(j) r)
    done;
    h
  in

  Random.self_init ();

  let swap i j =
    let (pi,pj) = (p.(i),p.(j)) in
    p.(i) <- pj;
    p.(j) <- pi
  in

  for i=0 to n-2 do
    let r = Random.int (n-i) in
    swap i (i+r)
  done;
  let rec loop h i r =
    (* already built the table for points 0...i, and they have dist r *)

    if i=n-1 then r else
      let i = i+1 in
      let (ix,iy) = boxify p.(i) r in
      let li = ref [] in
      for x = ix-1 to ix+1 do
        for y = iy-1 to iy+1 do
          li := (getbox h (x,y)) @ !li
        done
      done;

      let r' = List.fold_left (
        fun ac j -> min (dist i j) ac
      ) max_float !li in

      if r' < r then (
        loop (make_grid i r') i r'
      ) else (
        add_to_h h i (ix,iy);
        loop h i r
      )
  in

  let r0 = dist 0 1 in
  loop (make_grid 1 r0) 1 r0

let () =
  let p = [|(0.0,0.0); (3.0,4.0); (20.0,15.0); (15.0,20.0)|] in
  let answer = closest_pair p in
  Printf.printf "%f\n" answer;
```