In this lecture, we will see how the technique of *randomized incremental algorithms* can be applied to solve problems in computational geometry. We previously saw this same technique applied to linear programming with Seidel's 2D LP algorithm. We will once again use backwards analysis to prove bounds on the expected running time of these algorithms.

Objectives of this lecture

In this lecture, we will

- see how randomized incremental algorithms can be used for computation geometry
- give an expected linear-time algorithm for the *closest pair* problem
- give an expected linear-time algorithm for the *smallest enclosing circle* problem
- analyze these algorithms using *backwards analysis*

1 Model and assumptions

In this lecture we make the following assumptions. Like last time, we want to operate on real numbers, but we are now assuming a little bit more than last time.

- 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 in $O(1)$ time.
- We also assume that hashing is $O(1)$ time in expectation.

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

2 The closest pair problem

Problem: Closest pair

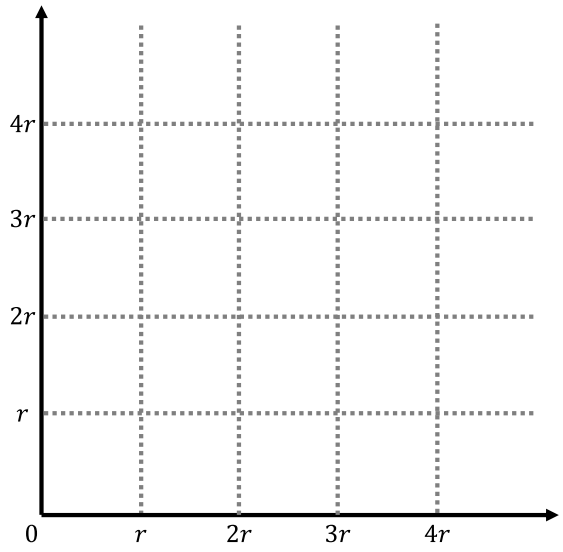
For any set of points P, let $\text{CP}(P)$ be the closest pair distance in P, i.e.,

$$\text{CP}(P) = \min_{\substack{p,q \in P \\ p \neq q}} \|p - q\|$$

Given a point set P, our goal is to compute $\text{CP}(P)$.

The naive algorithm of trying all pairs of points will cost $O(n^2)$ time. Our goal today is to come up with a faster algorithm. We will see that with some nifty randomization, we can actually achieve linear time!

A "grid" data structure We're going to define a "grid" data structure, and an API for it. The grid (denoted G) stores a set of points (that we'll call P) inside square cells of size $r \times r$. The number r is called "the grid size of G". The point (x, y) therefore goes in the grid cell $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$.



Now how does this thing help us find the closest pair of points? Well, intuitively, if we choose the grid size large enough, then the closest pair of points should end up in either the same grid cell, or in neighboring grid cells. Of course, if the grid size is *too large*, then there will be too many points in some cells, and finding the closest pair of points in a cell will take $O(n^2)$ time again...

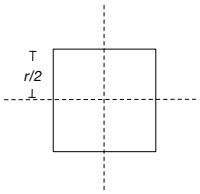
So, our goal is to somehow figure out just the right grid size such that it is small enough for there to be very few points per cell, but large enough such that the closest pair of points are in nearby cells. If we can do that, we'll find an efficient algorithm.

Choosing the right grid size Lets first focus on the criteria that we want the closest pair of points to live in the same or in neighboring grid cells. If the grid size is much smaller than the distance between the closest pair of points, then they might end up being very far away, so intuitively we want to choose the the grid size $r \approx \text{CP}(P)$. What if we just choose exactly $r = \text{CP}(P)$? I claim that this has all of the nice properties we want.

Claim 1: The right grid size

Given a grid G with points P and grid size $r = \text{CP}(P)$, no cell contains more than four points.

Proof. Imagine splitting a given cell into four $r/2$-sized sub-boxes.



The diameter of each sub-box is $\sqrt{2}r/2 < r$, so none of these sub-boxes can have more than one point in it, or that would imply that there exists a pair of points whose distance is less than r, which would contradict that $r = \text{CP}(P)$. Therefore there are at most four points in each cell. □

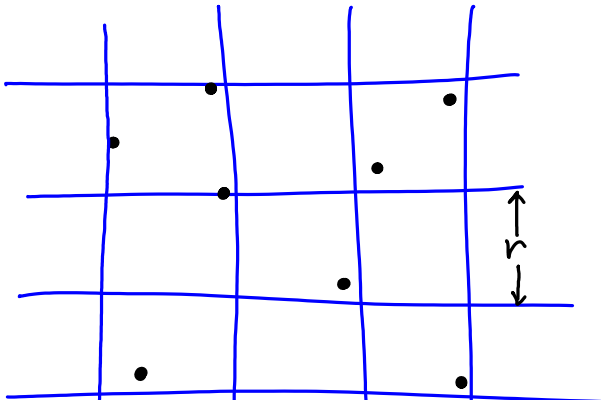
Okay great, so the perfect grid size is just to pick $r = \text{CP}(P)$, and then we're good! Oh wait... $\text{CP}(P)$ is exactly what we were trying to compute in the first place... What can we do? Lets try to solve the problem *incrementally*.

2.1 An incremental approach

We will start with just two points p_1 and p_2 and put them in a grid with $r = \|p - q\|$. Then we will continuously add the rest of the points into the grid, and if we violate the invariant that r is equal to the closest pair distance, we will just throw away the old grid and build a new one from scratch. Lets go into a bit more detail for each of these operations. Our grid will support the following API:

- MakeGrid(p, q): Make and return a new grid containing points p and q using $r = \|p - q\|$ (the distance between p and q) 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 $r' \geq r$. Essentially, the lookup function detects whether we have found a new closest pair.
- 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 new grid size.

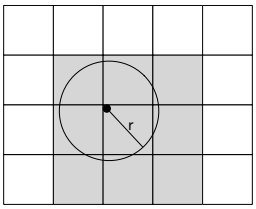
Here's an example Grid data structure satisfying the invariant.



We now discuss how to implement this API. Note that the grid could have arbitrarily many cells depending on the values of the point coordinates, so we can not afford to store them all explicitly. However, the vast majority of them will be empty, so we can store the grid using a dictionary data structure, powered by a hashtable to get $O(1)$-time operations. Specifically, we will maintain a hashtable whose keys are integer pairs (i, j), representing grid cells in the grid. Given a point (x, y), it belongs to the cell with key $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$.

MakeGrid(p, q) is simple. Just create a blank table, set $r = \|p - q\|$, and insert p and q .

Lookup(G, p) first computes the cell (i, j) where p belongs. It then looks in that cell, and the 8 surrounding ones, and computes the distance between p and every point in those neighboring cells. Call the smallest found distance r'. If $r' < r$, then we have found a new closest pair, so we return r'. Otherwise, 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 cells 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. This is because a cell contains at most 4 points by Claim 1.

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 the correct cell (i, j). This is correct, since it means that p does not create a new closest pair, so the grid size should be unchanged. This is $O(1)$ time. On the other hand if the Lookup() returns $r' < r$, then the algorithm has to throw away the current grid and start from scratch to build a new grid with size r'. This takes $O(i)$ time if there are i points now being stored in the data structure.

These algorithms are correct because they maintain the invariant that the grid size r is always equal to the closest pair distance.

Runtime analysis In the worst case, every newly inserted point might cause the closest pair to change, and hence require a regrid, so the runtime is

$$\sum_{i=1}^{n} i = \Theta(n^2).$$

Unfortunately this is no better than the brute-force approach! But how likely is it that we get so unlucky to have the closest pair change every iteration? What can we do to make this unlikely for any possible input?

2.2 Sariel Har-Peled's randomized $O(n)$ algorithm for closest pair

We can use the same approach as Seidel's 2D LP algorithm! Lets *randomly shuffle* the input points, then run the above algorithm. The claim is that by randomly shuffling, the probability that an insertion causes the closest pair to change is low.

Algorithm: Randomized incremental closest pair

```
RandomizedClosestPair(P) {
   Randomly permute the points. Call the new ordering p_1, p_2, ..., p_n.
   G = MakeGrid(p_1, p_2)
   for i = 3 to n do {
      r = Insert(G, p_i)
   }
   return r
}
```

Claim: Randomized incremental closest pair is fast

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.

We use an argument similar to the one we used for Seidel's LP algorithm, called "backward analysis":

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 $2/i$: if there is just one pair of points that achieves the minimum distance, then you have to remove one of those two points to increase the minimum. If there is more than one pair, then 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 runtime of the i^{th} insertion is

$$\underbrace{\frac{i-2}{i} \times O(1)}_{\text{Does not change}} + \underbrace{\frac{2}{i} \times O(i)}_{\text{Does change}} = O(1).$$

Thus the expected cost of the entire algorithm is $O(n)$ by linearity of expectation. □

3 The smallest enclosing circle problem

Problem: Smallest enclosing circle

Given $n \geq 2$ points in two dimensions, find the smallest circle (by radius) that contains all of the points.

The smallest enclosing circle is kind of like the convex hull. We are looking for a body that encloses all of the given points, except that this time, instead of searching for a polygon, we are looking for a circle. For notation purposes, lets define $\text{SEC}(p_1, \ldots, p_n)$ to be the smallest enclosing circle of the points $p_1, \ldots, p_n$.

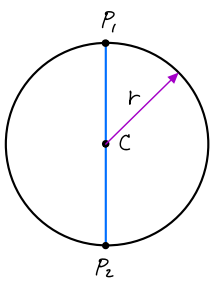
As usual, we will sweep edge cases under the rug by assuming that there are no sets of three colinear points in our input. Before we dive right into the algorithm, lets look at some useful properties of circles.

3.1 Base cases and useful properties of circles

Two points Suppose we start with exactly two points p_1 and p_2. There are infinitely many possible circles that enclose p_1 and p_2, and our algorithm can not try an infinite number of things, but we can restrict ourselves to a more reasonable set of possibilities.

Here's an idea that might seem obvious when we're just considering two points, but turns out to be one of the most powerful and useful observations when deriving computational geometry algorithms: pick a circle that touches the points. If I give you a circle that encloses the two points but doesn't touch them, then it can't be the SEC because I could shrink it and it will still contain the points.

How many circles are there that touch the points p_1 and p_2? Unfortunately there's still infinitely many of them, but there is a unique *smallest* circle that contains them, where p_1 and p_2 are on a diameter:



$$c = \frac{1}{2}(p_1 + p_2)$$

$$r = \frac{1}{2}|p_1 - p_2|$$

Three points Three points is where things get interesting. In the case of two, there were infinitely many possible circles that touch them, but it turns out that for three non colinear points, there exists a single unique circle that touches them.

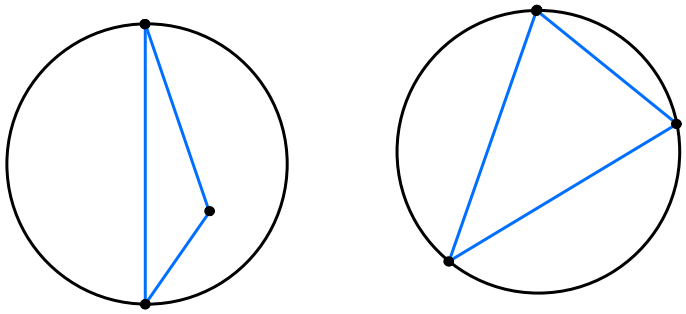
Claim 2: Three points make a circle

Given three non-colinear points, there exists a unique circle that touches them.

Exercise

Prove the claim above, that there is a unique circle that inscribes three non-colinear points.

Is this point always the smallest enclosing circle of the three points though? It might not be. There are two cases, either the triangle is acute, and the smallest enclosing circle will touch all three points, or it is obtuse, and the smallest enclosing circle will just touch two of them.



We can just try both cases, which is a constant number of choices, and we will have the smallest enclosing circle for three points.

3.2 The general case

Now we come to the general case, given $n > 3$ points, how do we find the smallest enclosing circle? Like before, we are faced with infinity many possible circles to choose from, so even if we wanted to implement a brute-force approach, it is not clear how to. We need to somehow reduce our infinite set of choices to a finite one... To do so, we will prove the following claim:

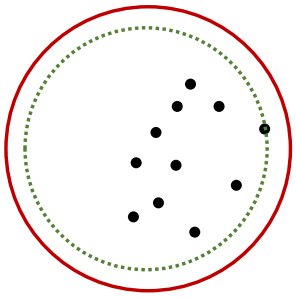
Claim 3: Three points defines the smallest enclosing circle

For any set of points, $\text{SEC}(p_1, \ldots, p_n)$ either touches two points p_i and p_j at opposite ends of a diameter, or touches three points p_i, p_j, p_k that form an acute triangle.

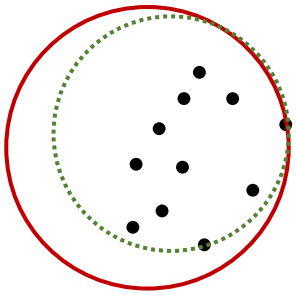
In other words, there exists i, j, k such that $\text{SEC}(p_1, \ldots, p_n) = \text{SEC}(p_i, p_j, p_k)$.

Proof. We'll consider a few cases:

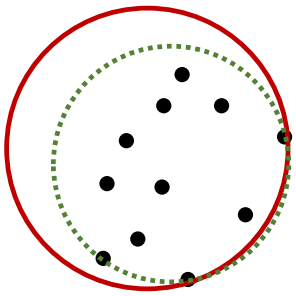
1. **No points:** Suppose I give you an enclosing circle (a circle that contains all of the points) but it doesn't touch any of them. Then I can shrink the circle by some ε until it does touch something and it still contains all the points, so it can't be the SEC.



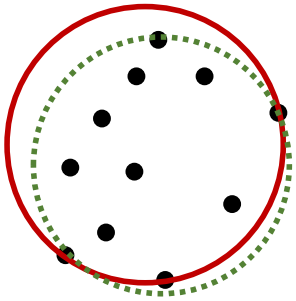
2. **One point:** Suppose I give you an enclosing circle that touches just one point. Then I can shrink the circle by some ϵ and translate it so that it still touches that one point, so it can't be the SEC.



3. **Two points:** Suppose an enclosing circle touches two points and these points are on opposite ends of a diameter. Then this is a lower bound on the size of any enclosing circle since it must contain these points, hence the circle is optimal. Suppose instead that the two points are not on opposite ends of a diameter, i.e., there is a greater than 180 degree gap between them. Then once again, we can shrink the circle by some ε and translate it.



4. **Three or more points:** If the circle touches at least three points but none of them form an acute triangle, we can again shrink the circle and translate it.



Therefore an SEC must either touch two points at a diameter, or touch three points of an acute triangle. □

What this claim is saying is that the smallest enclosing circle is determined by just some subset of three of the points in the input! So we do not have to try infinitely many possible circles to find the SEC, we have reduced to a finite number of possibilities. This gives us an $O(n^3)$-time brute-force algorithm: just try every triple of points and find the smallest enclosing circle, then return the largest one.

Note that we return the **largest one**, not the smallest one, because the smallest one might not contain all of the other points, but the largest one is guaranteed to, by Claim 3.

3.3 Beating brute force: an incremental algorithm

Continuing with the theme of this lecture, lets try to develop an incremental algorithm for the problem. That means we will start with just two points and find their SEC. Then, one by one we will add in another point and check whether it is contained within the current circle. If it is, we are good to continue. If it is not, we must find the new smallest enclosing circle of all the points. If we are lucky (or clever), the algorithm won't need to do the slow case very often.

Building an incremental algorithm Suppose we've computed the SEC of $p_1, \ldots, p_{i-1}$. How do we add the next point p_i? We have to consider two cases:

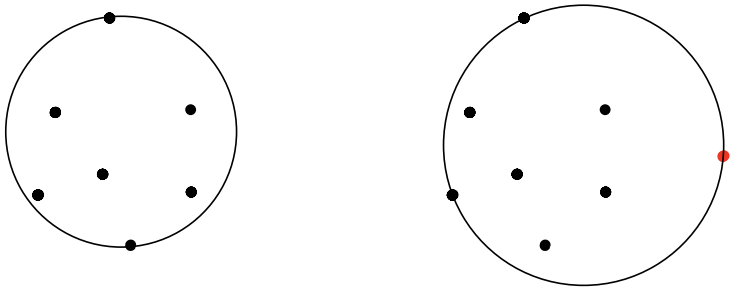
1. **Case 1:** p_i is inside SEC$(p_1, \ldots, p_{i-1})$: In this case, we don't have to do anything because the circle still contains all of the points, and adding a new point can not make the SEC smaller.
2. **Case 2:** p_i is not inside SEC$(p_1, \ldots, p_{i-1})$: This is the hard case. We could just decide to throw away the current circle and find the new SEC from scratch with brute force, but this would end up being really slow, even with the incremental approach, so we're going to need something more sophisticated.

Here's an observation that will help:

Claim: A new point locks in the new SEC

For a set of points $p_1, \ldots, p_i$, if SEC$(p_1, \ldots, p_{i-1}) \neq$ SEC$(p_1, \ldots, p_i)$, then p_i is on the boundary of the new SEC.

Proof. If p_i were not on the boundary, then by Claim 3, the SEC is determined by either two points along a diameter, or three points, but these points were all in the set before p_i was added, so the SEC shouldn't have changed. □



This allows us to improve on using brute force. If we insert a new point p_i and discover that it is outside the current SEC, we don't need to completely go from scratch and spend $O(i^3)$ time, we could lock p_i in and just do $O(i^2)$ work to find the other two points. In fact, we will see momentarily that we can do even better. To make this concrete, lets define a subroutine SEC1, which takes two parameters, a list of points $p_1, \ldots p_{i-1}$, and a point p_i, and finds the smallest enclosing circle of $p_1, \ldots, p_i$, but using the knowledge that p_i is definitely on the boundary. So, our algorithm for SEC currently looks like

```
SEC([p1, p2, ..., pn]) = {
   let C be the SEC of p1 and p2 (formed by a diameter between p1 and p2)

   for i = 3 to n do {
      if pi is not inside C then C ← SEC1([p1, ..., pi−1], pi)
   }
   return C
}
```

Implementing SEC1 We could implement SEC1 by trying all $O(i^2)$ pairs of other points and taking the best SEC that results, but this would take $O(i^2)$ time. We can still do much better than this! How? Same exact idea as the outer SEC algorithm! Lets solve SEC1 incrementally as well. Suppose we are given SEC1($[p_1, \ldots, p_n], q$) to solve.

- We can start with a circle around p_1 and q, then incrementally add the remaining points, one by one.
- If a point p_i is outside the current SEC, then we have to rebuild a new SEC.
- However, from the same argument as before, if this happens, we know for sure that p_i is on the boundary of the new one. So, we now need to solve the problem of finding the SEC of a set of points with *two fixed points* p_i and q. Hmm... time for another layer of indirection!

```
SEC1([p1, p2, ..., pk], q) = {
   let C be the SEC of p1 and q (formed by a diameter between p1 and q)

   for i = 2 to k do {
      if pi is not inside C then C ← SEC2([p1, ..., pi−1], pi, q)
   }
   return C
}
```

Here, SEC2($[p_1, \ldots p_k], q_1, q_2$) computes a smallest enclosing circle of $\{p_1, \ldots p_k, q_1, q_2\}$ with the knowledge that q_1 and q_2 must be on the boundary.

Implementing SEC2 Continuing the pattern, the last level of indirection is the simplest. We have two given points q_1, q_2 that are forced to be on the boundary, so we just need to loop through the k given points and pick the best third point. No more sophisticated tricks needed at this point!

```
SEC2([p1, p2, ..., pk], q1, q2) = {
   let C be the SEC of q1 and q2 (formed by a diameter between q1 and q2)

   for i = 1 to k do {
      if pi is not inside C then C ← Circle that touches (pi, q1, q2)
   }
   return C
}
```

Runtime analysis SEC2 always takes exactly $O(k)$ time no matter what. For SEC1, in the worst case, every new point triggers the need to find a new SEC, so SEC1 takes

$$\sum_{i=1}^{k} i = \Theta(k^2)$$

time. Similarly, in the worst case, SEC needs to find a new circle every iteration, so it takes

$$\sum_{i=1}^{n} i^2 = \Theta(n^3)$$

time. This is again, no better than brute force. Now its time to fix the problem with our favorite technique of the day, randomization!

3.4 Saving the day with randomization (again)

Lets add one additional line of code to SEC and SEC1 that randomly shuffles the points p, and then proceeds exactly as written. The almost-too-good-to-believe claim is that this brings us right down to linear time!

Claim: Randomized incremental smallest enclosing circle is linear time

SEC1 runs in $O(k)$ expected time, and SEC runs in $O(n)$ expected time.

Proof. Since SEC2 uses no randomization, it still always runs in $O(k)$ time deterministically. To analyze SEC1 and SEC, we once again apply *backward analysis*. Recall from Claim 3 that two or three points determine the SEC of the whole point set. So if we delete a point randomly among i points, the probability that the SEC changes is at most $\frac{3}{i}$. Therefore, the expected cost of one step of SEC1 is

$$\frac{3}{i} \times O(i) + \frac{i-3}{i} \times O(1) = O(1).$$

By linearity of expectation, the total expected cost of SEC1 is $O(k)$. By the same reasoning, the expected cost of one step of SEC is $O(1)$, and by linearity of expectation, the total expected cost is $O(n)$ □

Remark: Fun fact: less randomness suffices

In the algorithm described, we randomly permuted the points p in SEC and SEC1. It turns out that a single random permutation at the beginning of SEC is enough, and we don't actually need to permute inside SEC1. The proof becomes more difficult though because we don't have independence between different calls of SEC1 anymore, so we won't prove it.