CS 31: Algorithms (Spring 2019): Lecture 5

Date: 4th April, 2019

Topic: Divide and Conquer 3: Closest Pair of Points on a Plane

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please email errors to deeparnab@dartmouth.edu.

---

1 Closest Pair of Points on the Plane

We look at a simple geometric problem: given n points on a plane, find the pair which is closest to each other. More precisely, the n points are described as their (x, y) coordinates; point p_i will have coordinates (x_i, y_i). The distance between two points p_i and p_j is defined as

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

One could also look at other distances such as $d(p_i, p_j) = \max(|x_i - x_j|, |y_i - y_j|)$ and $d(p_i, p_j) = |x_i - x_j| + |y_i - y_j|$. What we describe below works (and is in one case easier) for both these as well.

CLOSEST PAIR OF POINTS ON THE PLANE
Input: n points $P = \{p_1, \ldots, p_n\}$ where $p_i = (x_i, y_i)$.
Output: The pair p_i, p_j with smallest $d(p_i, p_j)$.
Size: The number of points, n.

Once again, as many of the examples before, there is a trivial $O(n^2)$ time algorithm: simply try all pairs and return the closest pair. This is the naive benchmark which we will try to beat using Divide-and-Conquer.

How should we divide this set of points into two halves? To do so, let us think whether is there a natural ordering of these points? A moment's thought leads us to two natural orderings: one sorted using their x-coordinates, and one using their y-coordinates. Let us use $P_x[1:n]$ to denote the permutation of the n points such that $\mathsf{xcoor}(P_x[i]) < \mathsf{xcoor}(P_x[j])$ for $i < j$. Similarly we define $P_y[1:n]$. Getting these permutations from the input takes $O(n \log n)$ time.

Before moving further, we point out something which we will use later. Let $S \subseteq P$ be an arbitrary set of points of size s. Suppose we want the arrays $S_x[1:s]$ and $S_y[1:s]$ which are permutations of S ordered according to their xcoor's and ycoor's, respectively. If S is given as a "bit-array" with a 1 in position i if point $p_i \in S$, then to obtain S_x and S_y we don't need to sort again, but can obtain these from P_x and P_y. This is obtained by "masking" S with P_x; we traverse P_x from left-to-right and pick the point $p = P_x[i]$ if and only if $S[p]$ evaluates to 1. Note this is a $O(n)$ time procedure. This "dynamic sorting" was something we encountered in the Counting Inversions problem and is an useful thing to

know. More details can be found in the "Supplemental Problems: Sorting, Searching, and Stuff" file on Canvas. Anyway, back to our problem.

Given P_x, we can divide the set P into two halves as follows. Let $m = \lfloor n/2 \rfloor$ and $x^* := \text{xcoor}(P_x[m])$ be the median of P_x. Define $Q_x := P_x[1 : m]$ and $R_x := P_x[m + 1 : n]$, and let us use Q and R to denote the set of these point. The figure below illustrates this.

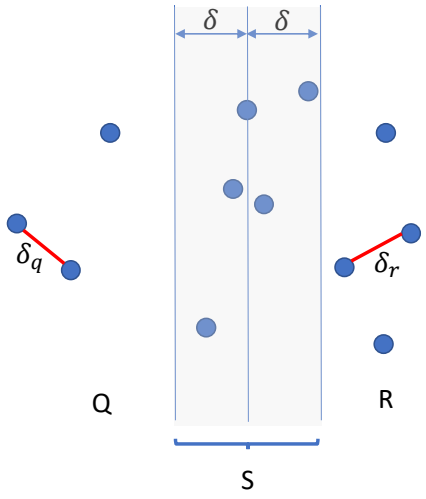

Figure 1: Closest pair in a plane

We recursively call the algorithm on the sets Q and R. Let (q_i, q_j) and (r_i, r_j) be the pairs returned. We will use[1] $\delta_q := d(q_i, q_j)$ and $\delta_r := d(r_i, r_j)$. Clearly these are candidate points for closest pair of points among P.

The other candidate pairs of P are precisely the *cross pairs*: (q_i, r_j) for $q_i \in Q$ and $r_j \in R$. Therefore, to conquer we need to find the nearest cross pair. Can we do this in much better than $O(n^2)$ time? If you think for a little bit, this doesn't seem any easier at all – can we still get a win? Indeed we will, but we need to exploit the **geometry** of the problem.

First let us note that we don't need to consider all pairs in $Q \times R$. Define $\delta := \min(\delta_q, \delta_r)$. Since we are looking for the closest pair of points, we don't need to look at cross-pairs which are more than δ apart.

Claim 1. Consider any point $q_i \in Q$ with $\text{xcoor}(q_i) < x^* - \delta$. We don't need to consider any (q_i, r_j) point for $r_j \in R$ as a candidate. Similarly, for any point $r_j \in R$ with $\text{xcoor}(r_j) > x^* + \delta$, we don't need to consider any (q_i, r_j) point for $q_i \in Q$ as a candidate.

[1] We haven't discussed the base case: if $n = 2$, then we return that pair; if $n = 1$, then we actually return $\perp$ and the corresponding $\delta = \infty$.

Proof. Any candidate (q_i, r_j) we need to consider better have $d(q_i, r_j) \leq \delta$. But

$$d(q_i, r_j) \geq |\mathsf{xcoor}(q_i) - \mathsf{xcoor}(r_j)|$$

Therefore, if $\mathsf{xcoor}(q_i) < x^* - \delta$, and since $\mathsf{xcoor}(r_j) \geq x^*$ for all $r_j \in R$, we get $|\mathsf{xcoor}(q_i) - \mathsf{xcoor}(r_j)| > \delta$. Thus, we can rule out (q_i, r_j) for all $r_j \in R$. The other statement follows analogously. □

Motivated by the above, let us define $Q' := \{q_i \in Q : \mathsf{xcoor}(q_i) \geq x^* - \delta\}$ and $R' := \{r_j \in R : \mathsf{xcoor}(r_j) \leq x^* + \delta\}$. That is $S = Q' \cup R'$ lies in the band illustrated in Figure 1. To summarize, we only need to look for cross-pairs in S. How will we go over all the cross pairs? Naively, we will pick a point $q \in S$ and go over all other $r \in S$ evaluating $d(q, r)$ as we go and store the minimum; then we repeat this for all $q \in S$ and take the smallest of all these minimums.

Once again, we want to use the observation that pairs which are $> \delta$ far needn't be considered. For a fixed $q \in S$, therefore, if we focused on the points $r \in S$ with $|\mathsf{ycoor}(r) - \mathsf{ycoor}(q)| \leq \delta$, it would suffice. We can do this using the sorted array P_y. To formalize this, first note that, as mentioned before, we can use P_y to find the array S_y which is the points in S sorted according to the ycoor's. To find the closest cross-pair, we consider the points in the increasing ycoor order; for a point $q \in S$ we look at the other points r subsequent to it in S_y having $\mathsf{ycoor}(r) \leq \mathsf{ycoor}(q) + \delta$, store the distances $d(q, r)$, and return the minimum. The following piece of pseudocode formalizes this.

```
1:  procedure ClosestCrossPairs(S, δ):
2:      ▷ Returns the cross pair (q, r) ∈ S × S with d(q, r) ≤ δ smallest
3:      We use P_y to compute S_y, that is, the points in S in sorted order.
4:      t ← ⊥ ▷ t is a tuple which will contain the closest cross pair
5:      dmin ← δ ▷ dmin is the current min init to δ
6:      for 1 ≤ i ≤ |S| do:
7:          p_cur ← S_y[i].
8:          ▷ Next, check if there is a point q_cur such that its distance to p_cur is < dmin.
9:          ▷ If so, then we define this pair to be t and define this distance to be the new dmin.
10:         ▷ Crucially, We don't need to check points which are δ away in the y-coordinate.
11:         j ← 1; q_cur ← S_y[i + j].
12:         while ycoor(q_cur) < ycoor(p_cur) + δ do:
13:             if d(p_cur, q_cur) < dmin then:▷ Modify dmin and t.
14:                 dmin ← d(p_cur, q_cur);
15:                 t ← (p_cur, q_cur)
16:             j ← j + 1; q_cur ← S_y[i + j]. ▷ Move to the next point in S_y.
17:     return t ▷ Could be ⊥ as well.
```

Remark: *One may wonder that we are not returning cross-pairs as we could return q, r both in Q'. However, for any pair (q, r) returned, we have $d(q, r) < \delta$; since $\delta = \min(\delta_q, \delta_r)$, this pair can't lie on the same side.*

Armed with the above "conquering" step, we can state the full algorithm.

```
1:  procedure ClosestPair(P):
2:      ▷ We assume n = |P|.
3:      ▷ We assume arrays P_x[1 : n] and P_y[1 : n] which are xcoor and ycoor-sorted P.
4:      if n ∈ {1, 2} then:
5:          If n = 1 return ⊥; else return P.
6:      m ← ⌊n/2⌋
7:      Q be the points in P_x[1 : m]
8:      R be the points in P_x[m + 1 : n]
9:      (q1, q2) ← ClosestPair(Q); δ_q ← d(q1, q2).
10:     (r1, r2) ← ClosestPair(R); δ_r ← d(r1, r2).
11:     δ ← min(δ_q, δ_r)
12:     x* ← xcoor(P_x[m]).
13:     Compute S ← {p_i : x* − δ ≤ xcoor(p_i) ≤ x* + δ}. ▷ Store as indicator bit-array
14:     ▷ All cross-pairs worthy of consideration lie in S
15:     (s1, s2) ← ClosestCrossPair(S)
16:     return Best of (q1, q2), (r1, r2) and (s1, s2).
```

How long does the above algorithm take? Note $|S|$ could be as large as $\Theta(n)$. The inner while loop, a priori, can take $O(|S|)$ time, and thus along with the for-loop, the above seems to take $O(n^2)$ time. Doesn't seem we have gained anything. Next comes the real geometric help.

Remark: *In class, we looked at a much better lemma than before with 72 replaced by 8. Still, I think the arguments below has a certain generality which is good to know. When arguing about a set of points which are further apart, it is a good idea to consider small balls (which are circles in two dimensions) around them, and arguing. Also the argument below can be used for solving closest pair in higher dimensions; something which I may post notes about in a later date.*

Lemma 1. Fix any point $q \in S$. Then there are at most 72 points $r \in S$ with $d(q, r) \leq \sqrt{5}\delta$.

Before we prove this, let us first see why is this lemma useful.

Corollary 1. The inner while loop always takes $O(1)$ time.

Proof. Suppose not, that is, the while loop runs for > 72 iterations for some $q = S_y[i]$. Then, there are at least 72 points $r \in S$ s.t. $\mathsf{ycoor}(r) \leq \mathsf{ycoor}(q) + \delta$, or $|\mathsf{ycoor}(r) - \mathsf{ycoor}(q)| \leq \delta$. Since $q, r \in S$, we know that $|\mathsf{xcoor}(r) - \mathsf{xcoor}(q)| \leq 2\delta$. This means that $d(q, r) \leq \sqrt{5}\delta$. But this contradicts Lemma 1. □

Proof of Lemma 1. Before going over the math, let's see the intuition. Suppose there are > 72 points of S in a circle of radius $\sqrt{5}\delta$ around a point q. Now at least 36 of these points belong to one set Q or R; let's without loss of generality this is R. What do we know about these 36 points – their pairwise distances are $\geq \delta$. How can we have so many points (if 36 doesn't sound a lot, thing 36000) which are each δ-far from each other, all sitting in a circle of radius $\sqrt{5}\delta$? We can't : try to picture it. You will see lot of congestion.

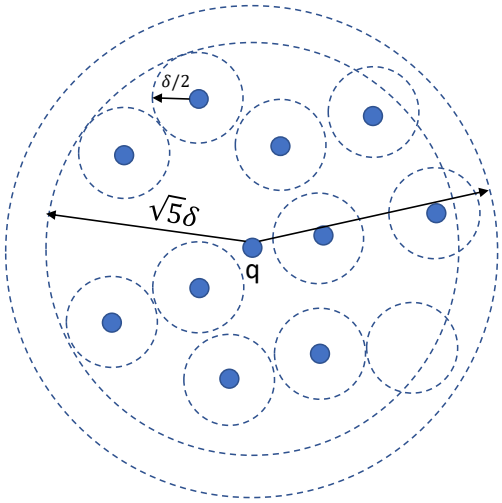

Figure 2: Small Non-overlapping circles inside another circle. Can't be many.

Now we do the math. Here is the formal argument. Let's take these 36 points of R and draw circles of radius $\delta/2$ around them. Since any two pair of points is $\geq \delta$, all these circles are non-overlapping. Furthermore, all these 36 circles lie in the bigger circle of radius $(\sqrt{5} + 1/2)\delta$ around q. See the Figure 2 for an illustration.

We get a contradiction by an "area" argument. The area of the big circle is $\pi \cdot \delta^2 \cdot (\sqrt{5} + 1/2)^2 < 9\pi\delta^2$. The area of each small circle is $\pi \cdot \delta^2/4$. Since the 36 small circles all fit in the big circle and they are *non-overlapping*, the sum of the areas of the small circles must be $\leq$ the area of the big circle. This is where we reach a contradiction – the 36 small circles have area $9\pi\delta^2$. □

If $T(n)$ is the worst case running time of CLOSESTPAIR when run on point set of n points, we get the recurrence inequality which I hope we all have learned to love:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

This evaluates to $T(n) = O(n \log n)$.

Theorem 1. The closest pair of points among n points in a plane can be found by CLOSESTPAIR in $O(n \log n)$ time.

2 Fibonacci Numbers: Recursion with Memory

We are done with Divide-and-Conquer for this class. Next topic: dynamic programming (DP). Dynamic Programming is an essential tool in the algorithm designer's repertoire. It solves problems that a first glance seems to suggest are terribly difficult to solve. It is, unfortunately, also something that students have trouble understanding and using. But it *is* a very simple idea, and once one gets used to it, hard to muck up. We will begin DPs in earnest from next class, but today we explore the main idea behind dynamic programming: *recursing with memory* or *bottom-up recursion* or *smart recursion*.

Let us recall Fibonacci numbers.

$$F_1 = 1, F_2 = 1, \quad \forall n > 2, F_n = F_{n-1} + F_{n-2} \tag{1}$$

Here is a simple computational problem.

FIBONACCI
Input: A number n.
Output: The nth Fibonacci number, F_n.
Size: n.

Remark: *"Now hold on," I hear you cry, "we are back to handling numbers as input. Then if the input is n, shouldn't we consider the number of bits in n, that is $\log n$, as the size and not n?". Perfectly valid question. The answer lies in the output. The exercise below shows that the number of bits required to write F_n is $\Theta(n)$. This is the reason we are going to take n as the size.*

But I am *going to cheat a below – when I add and multiply numbers, I am still going to* wrongly *assume they are $\Theta(1)$ time operations. In fact, they may take $\Theta(n)$ time. You should fill in the gaps.*

✍

Exercise: *Prove that for any n, we have $2^{n/2} \leq F_n \leq 2^n$.*

The definition (1) of Fibonacci numbers implies the following recursive algorithm.

```
1: procedure NaiveFib(n):
2:     if n ∈ {1, 2} then:
3:         return 1
4:     else:
5:         return NaiveFib(n − 1) + NaiveFib(n − 2)
```

We will see the above is a disastrous thing to do. However, for all the problems we will encounter for Dynamic Programming, if you have obtained the disastrous algorithm as above, you are probably close to victory. That is, what we are going to see below to smartly implement the recursion in (1), will also perhaps work for the recursion you have obtained to get your "disastrous algorithm".

What is the running time of NAIVEFIB? As warned before, I am assuming (wrongly) that the addition in Line 5 takes $\Theta(1)$ time. We see that the recurrence which governs the running time is

$$T(n) \leq T(n-1) + T(n-2) + \Theta(1)$$

Even if we ignore the $\Theta(1)$ in the RHS above, we see that the recurrence governing $T(n)$ is eerily similar to (1). And that is not good news – it says $T(n)$ can be as large as F_n which we know from the exercise above is $\geq 2^{n/2}$. Yikes!

There are two ways to fix this. Both involve the same principle. We observe that the above recursive implementation is super wasteful. To see this, note the function called on $n = 8$ recursively called the function with $n = 7$ and $n = 6$. The recursion with $n = 7$ again calls the function with $n = 6$ again, thereby doing twice the work as required. The idea is: *if we remember solutions to smaller subproblems, then we don't have to re-solve them.*

Implementation via Memoization.

```
1: procedure MEMOFIB(n):
2:     Implement a "look-up table" T.
3:     Define T[1] ← 1; T[2] ← 1.
4:     if T[n] is defined then:
5:         return T[n]
6:     else:
7:         t ← MEMOFIB(n − 1) + MEMOFIB(n − 2)
8:         Set T[n] ← t
9:         return t
```

(We **did not** do this in class.) The *memoization* approach gets to the heart of the problem described above. It stores all previously computer Fibonacci numbers in a look-up table T. Therefore, the running time (assuming all look-ups and additions are $\Theta(1)$ time operations) takes $O(n)$ time.

Bottom-Up Implementation: The "Table" method. This is the method which makes computation more explicit and will be what we will use throughout the course. The method described below seems a little wasteful implementation of the memoization idea; it has the benefit of (hopefully) being clearer.

Remark: *Few comments: (a) most times, whatever can be solved using the bottom-up implementation can also be addressed by memoization, and (b) often the table method can be implemented as efficiently as memoization, (c) the table method wins when we are interested in a lot of answers and not just F_n; say all the prime-indexed Fibonacci numbers.*

There are many who feel memoization is more 'natural'; I don't. Perhaps it has got to the way my mental models are. In any case, if you do use memoization or bottom-up, the important thing to remember is to be correct.

We first observe that the *solution* to the problem FIB(n) depends on the *solutions* to the problems FIB($n-1$) and FIB($n-2$). Thus, if we stored these solutions in an array (sounds very much like a look-up table) $F[1:n]$, then the following picture shows the dependency of the various entries.

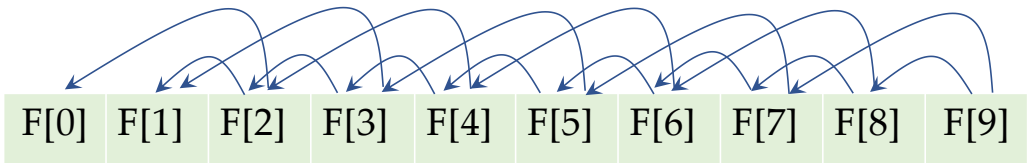

Figure 3: Dependency of the Fibonacci Array

Note two things: (a) The "graph" has no cycles. Otherwise, there will be a cyclic dependency and it doesn't make sense. (b) There are "sinks" in this graph: points from which no arrows come out. These are the *base cases*; $F[1]$ and $F[2]$ don't need any computation; they are both 1. Given this graph above, the algorithm to obtain the nth position $F[n]$ becomes clear – traverse it from the "sink" to $F[n]$. Here's how to do it.

```
1: procedure FIB(n):
2:     Allocate space F[1 : n]
3:     Set F[1] ← 1; F[2] ← 1.
4:     for i = 3 to n do:
5:         F[i] ← F[i − 1] + F[i − 2]
6:         ▷ Note: the computation of F[i] requires precisely the F[j]'s the F[i] points to
7:     return F[n]
8:     ▷ Note: we have actually found all the F[j] for j ≤ n. The Table method often does
       more work than needed.
```

2.1 Binomial Coefficients

Here is another example: *binomial coefficients*. Given non-negative integers $n, k \leq n$, one uses $\binom{n}{k}$ to denote the number of ways of choosing k distinct items from n distinct items.

The following is a well known recurrence *equality* (just like Fibonacci numbers).

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{2}$$

BINOMIAL
Input: Numbers $n, k \leq n$.
Output: The Binomial Coefficient $\binom{n}{k}$
Size: n.

✍

Exercise:*Prove the above equality.*

Once again, there is a naive recursive algorithm which the above recurrence (2) readily defines. I am not writing it below, but you should perhaps convince yourself of it.

We observe that to compute $\binom{n}{i}$ we need only $\binom{n-1}{i}$ and $\binom{n-1}{i-1}$. Thus if we have "smaller" binomial coefficients, then we can use them to get larger binomial coefficients. This suggests we store for all $1 \leq m \leq n$ and all $1 \leq j \leq k$, the binomial coefficients $\binom{m}{j}$; this we store in a two-dimensional table $B[m, j]$. Once we make this decision, the following code tells us how to "fill up the table".

```
1:  procedure BINOM(n, k):
2:      Allocate space B[0 : n, 0 : k].
3:      Set B[m, 0] = 1 for all m ▷ Base case of (m choose 0) = 1 for all m.
4:      Set B[m, j] = 0 for all j > m.▷ Base Case : there is zero ways of choosing a larger
        number of items from a smaller number of items.
5:      for m = 1 to n do:
6:          for j = 1 to k do:
7:              B[m, j] = B[m − 1, j] + B[m − 1, j − 1].
8:      ▷ Note: the computation of B[m, j] requires B[m − 1, j] and B[m − 1, j − 1] and
        they have been computed before.
9:      return B[n, k]
10:     ▷ Note: we have actually found all the B[m, j] for m ≤ n, j ≤ k. The Table method
        often does more work than needed.
```

The running time of the above is $O(nk)$ time.