CS 6363.003.21S Lecture 6—February 4, 2021

Main topics are #example/closest_pair and #example/Karatsuba_multiplication.

Closest Pair

- Today, we're going to look at one or two examples of divide-and-conquer that aren't yet another example of us doing sorting or order statistics
- Several of us professors at UTD focus on a particular area of algorithm design called *computation geometry*. So today, I'm going to show you how divide-and-conquer can help in solving geometric problems. In particular, we're going to focus on the problem of finding the *closest pair* of points in the plane
- We're given a set of n points in the plane. To be concrete, let's say they're given as a pair of arrays X[1 .. n] and Y[1 .. n] where the ith point has coordinates (X[i], Y[i]).
- Our goal is to find the two points that are closest. For simplicity, we'll focus today on computing their distance.
- Now, the most obvious thing we can do is explicitly check the distance between every pair of points. But that's Theta(n^2) pairs to check! Can we solve the problem more quickly?
- Let's try using divide-and-conquer. The first thing that comes to mind for me when doing divide-and-conquer for geometry is to partition the input as evenly as possible along one dimension, recursively solve the problem on the two subsets of the partition, and fill in any gaps that are leftover.
- So let's try that: we'll partition the points so half lie to the left of the median x-coordinate and the rest lie to the right.
- Then we'll ask the Recursion Fairy to find the closest pair distances in the left and right halves separately.
- But maybe the closest pair includes one point from the left half and one point from the right? We'd better check all those pairs as a conquer step and then return the smallest distance we found. Here's the pseudocode:

```
ClosestPair(X[1 .. n], Y[1 .. n]):
  if n ≤ 3
      solve by brute force

  XL[1 .. ⌊n/2⌋] and YL[1 .. ⌊n/2⌋] ← leftmost ⌊n/2⌋ points
  ℓ ← ClosestPair(XL[1 .. ⌊n/2⌋], YL[1 .. ⌊n/2⌋]) ⟨⟨Recurse!⟩⟩
  XR[1 .. ⌈n/2⌉] and YR[1 .. ⌈n/2⌉] ← rightmost ⌈n/2⌉ points
  r ← ClosestPair(XR[1 .. ⌈n/2⌉], YR[1 .. ⌈n/2⌉]) ⟨⟨Recurse!⟩⟩

  m ← ∞ ⟨⟨Find closest pair between two halves⟩⟩
  for i ← 1 to ⌊n/2⌋
      for j ← 1 to ⌈n/2⌉
          if Distance(XL[i], YL[i], XR[j], YR[j]) < m
              m ← Distance(XL[i], YL[i], XR[j], YR[j])

  return min{ℓ, r, m}
```

- So what is the running time here? We do two recursive calls on sets of half the size. But then we have double-nested for loops with around n/2 iterations each. That's still Theta(n^2) distance checks total!
- The recurrence itself comes out to be T(n) = 2T(n/2) + Theta(n^2). As you might guess by now, the recursion tree level sums are going to decrease exponentially. But we still have a Theta(n^2) time algorithm.
- So now what? Well, it turns out there are two observations we can make that will help us speed up this algorithm.
- Let d ← min{ell, r} be the smaller of the two distances returned by the Recursion Fairy.
- First off, suppose point p is more than d distance away from the vertical line through the median point by x-coordinate. p is also more than d distance away from *all* points on the other side of that line. So there's no reason to consider point p during our conquer step!
- If the closest pair uses a point on each side of the median line, then both of those points have to lie within distance d of that line.
- Great! But it's possible that most or even all of the points lie within distance d of the line.
- For the second observation, consider a point p on the left side of the median line, and suppose the closest pair includes p and a point q on the right side of the line.
- We already established that q lies within horizontal distance d of the median line.
- But it must also be true that q lies within *vertical* distance d of p. In other words, q must lie within a rectangle of width d and height 2d where the left side of the rectangle lies on the median line and the bottom and top are d units below and above p, respectively.
- So for each point p on the left, we need only check points within p's rectangle.
- And here's the kicker: The Recursion Fairy has already established that all pairs of points to the right of the line are distance at least d apart.
- It turns out we can only fit 6 points of pairwise distance ≥ d in a d x 2d rectangle, so we only need to compute distances between the n / 2 choices of p and at most 6 other points each.
- I'll prove a slightly weaker result that at most 8 points fit in the rectangle. Either way, that's a

constant number of comparisons for each point p on the left.
    - Divide the rectangle up into eight (d/2) x (d/2) squares.
    - No two points can fit in a single square, because they'd be distance at most (d/2) / sqrt{2} = d / sqrt{2} apart.
    - So there must be at most one point per square or 8 points total in the rectangle.
- OK, it's time to turn these observations into an efficient algorithm.
- We'll ask the Recursion Fairy to compute closest pair distances within the leftmost n /2 points and the rightmost n / 2 points as before. Then we take note of the smaller distance returned d and begin ignoring all points more than d distance from the median line.
- We'd like to spend time proportional to the number of pairs we're actually comparing during the conquer step.
- To do so, we'll loop over points p to the left of the line in increasing order by y-coordinate.
- And we'll also keep a finger on the lowest point on the right side that's still within vertical distance d of our current point p.
- For each point p, we'll move our finger up to the lowest point in p's rectangle (if necessary), and then perform comparisons with the other points in the rectangle. So as we're dealing with points on the left side, we're essentially scanning points on the right side in increasing order by y-coordinate.
- And one final trick: it's going to slow us down a bit if we have to sort all the points vertically every time we do a recursive call. So instead, we'll sort the points before the initial call to our algorithm and make sure to pass along the sorted subsequences every time we make a recursive call.
- Here's the final pseudocode:

```
ClosestPairFast(X[1 .. n], Y[1 .. n]):
  ⟨⟨Assumes points come pre-sorted by y-coordinate⟩⟩
  if n ≤ 3
      solve by brute force

  XL[1 .. ⌊n/2⌋] and YL[1 .. ⌊n/2⌋] ← leftmost ⌊n/2⌋ points
  ℓ ← ClosestPairFast(XL[1 .. ⌊n/2⌋], YL[1 .. ⌊n/2⌋]) ⟨⟨Recurse!⟩⟩
  XR[1 .. ⌈n/2⌉] and YR[1 .. ⌈n/2⌉] ← rightmost ⌈n/2⌉ points
  r ← ClosestPairFast(XR[1 .. ⌈n/2⌉], YR[1 .. ⌈n/2⌉]) ⟨⟨Recurse!⟩⟩
  d ← min{ℓ, r}

  ⟨⟨Find closest pair between two halves⟩⟩
  XL'[1 .. k] and YL'[1 .. k] ← subset of leftmost ⌊n/2⌋ points with x-coordinate ≥ XR[1] − d
  XR'[1 .. o] and YR'[1 .. o] ← subset of rightmost ⌈n/2⌉ points with x-coordinate ≤ XR[1] + d
  m ← ∞
  jmin ← 1
  for i ← 1 to k
      while jmin ≤ o and YR'[jmin] < YL'[i] − d
          jmin ← jmin + 1
      j ← jmin
      while j ≤ o and YR'[j] ≤ YL'[i] + d
          if Distance(XL'[i], YL'[i], XR'[j], YR'[j]) < m
              m ← Distance(XL'[i], YL'[i], XR'[j], YR'[j])
          j ← j + 1

  return min{ℓ, r, m}
```

- We're doing two recursive calls on point sets of half the size.
- Outside the recursive calls, we're looping over O(n) values of i, one for each left side point close to the median line. All updates to our lowest rectangle point (represented by jmin) take O(n) time total. And as argued earlier, we're doing at most 6 x n / 2 = O(n) distance computations in the second while loop.
- The runtime recurrence comes out to be T(n) = 2T(n/2) + O(n). Which, again, solves to O(n log n). That's much better than Theta(n^2)!

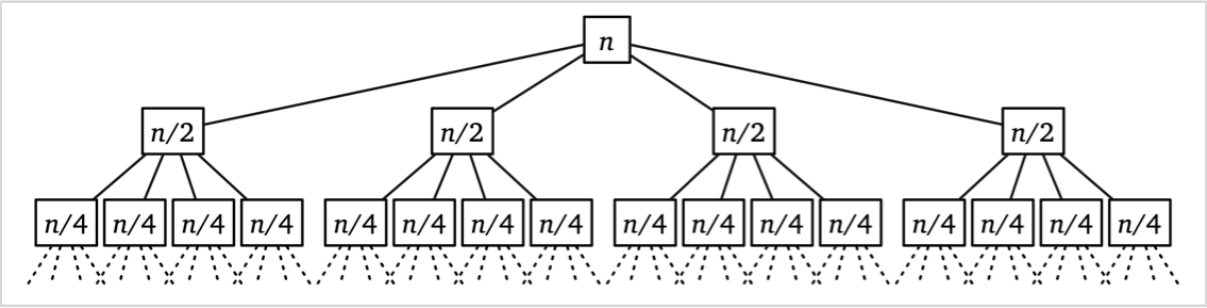
Multiplication

- And here's one last example in case we have time for it.
- A couple times already, we've discussed an algorithm for multiplying two large numbers x and y.
- But as we saw, that algorithm takes O(n^2) time to multiply two n-digit numbers.
- Maybe we can do better using divide-and-conquer?
- Let m be some non-negative integer. We can split the digits of x and y roughly in half so that x = (10^m a + b) and y = (10^m c + d) for some numbers a, b, c, and d.
- And now multiplying x and y comes down to observing (10^m a + b)(10^m c + d) = 10^(2m) ac + 10^m (bc + ad) + bd.
- The four products that don't involve 10^something use numbers with fewer digits, so maybe we can use divide-and-conquer!

- We'll do the easier multiplications recursively, and then combine them using that formula.
- In pseudocode, we get the following algorithm. SplitMultiply(x, y, n) computes x * y assuming they both use at most n digits.

```
SplitMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← SplitMultiply(a, c, m)
      f ← SplitMultiply(b, d, m)
      g ← SplitMultiply(b, c, m)
      h ← SplitMultiply(a, d, m)
      return 10^(2m) e + 10^m (g + h) + f
```

- Correctness follows easily from induction (if n = 1, we just return the product. Otherwise, we correctly multiply the smaller values by the induction hypothesis and combine them according to the identity.)
- So what is the running time? The mods and multiplying by 10^whatever takes linear time since it's just digit shifts. Between that and the additions, everything outside the recursive calls takes O(n) time. There are 4 recursive calls on problems of roughly half the size, so we'll say the running time follows the recurrence T(n) = 4T(n/2) + O(n).
- The recursion tree method shows us the level-sums are exponentially increasing, meaning T(n) is bounded by the number of leaves. T(n) = O(n^{log_2 4}) = O(n^2).

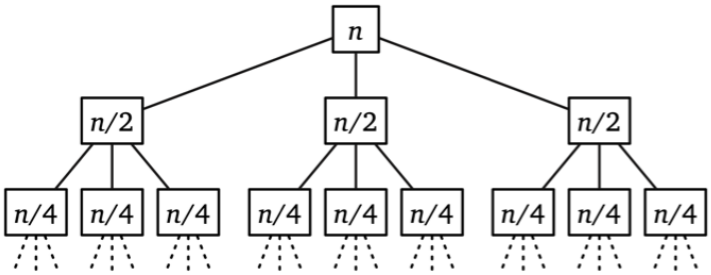


- Oh, that didn't help at all.
- In the 1950's, renounced mathematician Andrei Kolmogorov publicly conjectured that there is *no* algorithm for multiplying two n-digit numbers in o(n^2) time. He organized a seminar in 1960 where he planned to discuss this conjecture and several related problems. Almost one week later, 23-year-old student Anatolii Karatsuba found a better algorithm after all. Kolmogorov told the seminar participants about the better algorithm, and immediately terminated the seminar.
- So, how do we do better? Well, for our divide-and-conquer algorithm, we need to compute bc + ad. It turns out, given ac and bd, we can compute that sum using only *one* more multiplication instead of two.

- bc + ad = ac + bd - ac + bc + ad - bd = ac + bd - (a - b)(c - d).
- So we get this alternative algorithm with only three recursive calls instead of four!

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^(2m) e + 10^m (e + f − g) + f
```

- Now we have three recursive calls of size roughly n/2, so the running time follows T(n) = 3T(n/2) + O(n).
- The level-sums of the recursion tree still form an increasing geometric series bounded by the number of leaves,…



but now T(n) = O(n^{log_2 3}) ~ O(n^1.58496). That's a big improvement!

- So when you're designing your own divide-and-conquer algorithms, see if you can limit the number of recursive calls you perform to help speed things up.
- As for multiplication, you can take this idea even further by splitting the numbers into more pieces and combining the products in more complicated ways.
- And after a long line of improvements David Harvey and Joris van der Hoeven ended up finding an O(n log n) time algorithm using other techniques. This algorithm was accepted for publication only a couple months ago!