EECS 660: FUNDAMENTALS OF COMPUTER ALGORITHMS

MODULE VI: DIVIDE AND CONQUER, PART 2

DISCLAIMER

- This document is intended to be used for studying EECS 660 Data Structures, only by students who are currently enrolled in the course.

-

- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Algorithm Design*** 4th edition, by Jon Kleinberg and Eva Tardos and ***Analysis of Algorithms*** 2nd edition, by Jeffery McConnell.

OUTLINE

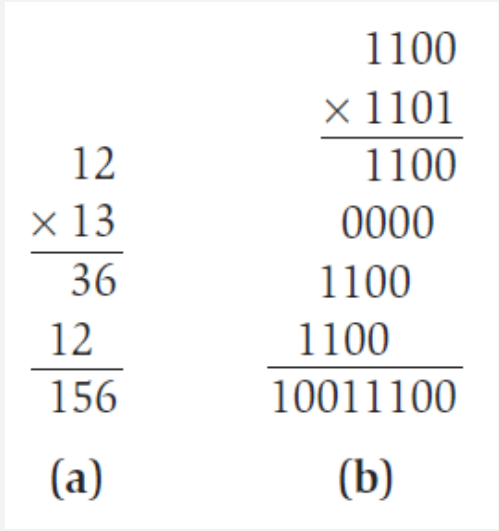
- Some more examples of divide and conquer algorithms
  - integer multiplication
  - counting inversions
  - closest pair of points

INTEGER MULTIPLICATION

- Imagine the multiplication of two large numbers
  - although the multiplication of two small numbers (without causing overflow) can be done in $O(1)$ time
  - a multiplication of two large numbers (with the consideration of overflow) cannot be done in $O(1)$ time

INTEGER MULTIPLICATION

- Consider we are multiplying two integers, each with n digits (if not, we can pad 0s to the left-hand side of the smaller number)
  - what we will be doing is similar to what we will do by hand: simply multiply each digit of the second number with each digit of the first number, and add up the results
  - clearly, it is easy to see that an multiplication of single digits takes $O(1)$ time
  - so, this naïve algorithm takes $O(n^2)$ time



INTEGER MULTIPLICATION

- Let's see how we can divide the multiplication into subproblems:
  - we could partition each number into two subsets of digits: half digits that are more significant and half digits that are less significant
  - say we are computing 1200*3612, we can partition the first number as 12 (we mark it red to indicate that it should be multiplied with 100) and 00, and the second number as 36 and 12
  - then we will be performing 4 multiplications: 12*36=432, 00*36=0, 12*12=144, 00*12=0
  - and we know if we have two red numbers multiplying, we should raise the result with *10000; and if we have one red number, we should raise the result with *100
  - so, finally we will have 432*10000+0*100+144*100+0=4334400 (and you can verify that by computing 1200*3612 with your calculator)

INTEGER MULTIPLICATION

- We can generalize this idea as the follows:
  - consider the multiplication of two integers x and y, each with n digits
  - we can make $x = x_0 2^{n/2} + x_1$, and $y = y_0 2^{n/2} + y_1$
  - then, $xy = \left(x_0 * 2^{\frac{n}{2}} + x_1\right) * \left(y_0 * 2^{\frac{n}{2}} + y_1\right) = x_0 y_0 2^n + (x_0 y_1 + x_1 y_0) 2^{\frac{n}{2}} + x_1 y_1$
  - through computing the four subproblems $x_0 y_0$, $x_0 y_1$, $x_1 y_0$, and $x_1 y_1$, we will eventually get the final product

INTEGER MULTIPLICATION

- We will skip the analysis of the algorithm's complexity since it is obvious.

- To analyze its time complexity, we see that the algorithm divides the larger problem into equal-sized subproblems; so, we can use the Master's theorem:
  - $a = 4$, since we need to compute 4 multiplications as subproblems
  - $b = 2$, the size of the subproblem is halved
  - $c = 1$, since adding the results of the 4 multiplications clearly takes linear time (the maximum number of digits the product of two n-digit numbers can have is $2n$; try 99*99)
  - using Master's theorem, we have the time-complexity expression as $O\left(n^{\log_b a}\right) = O\left(n^{\log_2 4}\right) = O(n^2)$, <u>which is exactly the same as the naïve algorithm!</u>

INTEGER MULTIPLICATION

- We shall further analyze the expression:
  - $xy = \left(x_0 * 2^{\frac{n}{2}} + x_1\right) * \left(y_0 * 2^{\frac{n}{2}} + y_1\right) = x_0 y_0 2^n + (x_0 y_1 + x_1 y_0) 2^{\frac{n}{2}} + x_1 y_1$
  - notice that we only need the sum of $x_0 y_1 + x_1 y_0$, but not necessarily $x_0 y_1$ nor $x_1 y_0$
  - is there a faster way to get $x_0 y_1 + x_1 y_0$?
  - note that $(x_0 + x_1)(y_0 + y_1) = x_0 y_0 + x_0 y_1 + x_1 y_0 + x_1 y_1$, so $x_0 y_1 + x_1 y_0 = (x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1$
  - we know that we need to compute $x_0 y_0$ and $x_1 y_1$ as subproblems; we will also compute $(x_0 + x_1)(y_0 + y_1)$ as the third subproblem
  - we can approximate the computation of $(x_0 + x_1)(y_0 + y_1)$ as a subproblem with size $n/2$, since the sum of two $n/2$-digit numbers can have at most $\frac{n}{2} + 1$ digits
  - and all additional additions (i.e., $x_0 + x_1$ and $y_0 + y_1$) and subtractions (i.e., $(x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1$) can be done in $O(n)$ time

INTEGER MULTIPLICATION

- Now, we can revise our analysis of the time complexity:
  - $a = 3$, since we need to compute 3 multiplications as subproblems
  - $b = 2$, the size of the subproblem is halved
  - $c = 1$, since adding the results of the 4 multiplications clearly takes linear time (the maximum number of digits the product of two n-digit numbers can have is $2n$; try 99*99)
  - using Master's theorem, we have the time-complexity expression as $O\left(n^{\log_b a}\right) = O\left(n^{\log_2 3}\right) = O(n^{1.59})$, which is faster than the naïve algorithm!

INTEGER MULTIPLICATION

- In theory, we can actually do better, by dividing each number into three subsets.
- The corresponding algorithm is called Toom-Cook algorithm, which have 5 subproblems with sizes $n/3$
  - the resulting time complexity is $O\left(n^{\log_b a}\right) = O\left(n^{\log_3 5}\right) = O(n^{1.46})$

- If interested, read it on Wikipedia https://en.wikipedia.org/wiki/Toom%E2%80%93Cook_multiplication
  - Toom-Cook algorithm will not be tested in the exams

COUNTING INVERSIONS

- Consider the following problem: we wish to study which courses in computer science are strongly tied to EECS 660 Algorithms.
  - we consider two courses are strongly tied if students do well in EECS560 also tend to do well in the other class
  - for example, students do well in EECS 660 also tend to do well in EECS 560 Data structure, but the tie could be weaker in, say, Physics classes
  - of course, we can quantify the similarity in statistical sense, such as using Pearson's or Spearman's correlation
  - but here, we would like to use another measure, called **rank inversions**

COUNTING INVERSIONS

- Say we have exactly the same set of students who have taken EECS 560 and EECS 660, and we have finished the grading with the students' rankings in the two courses.
  - for example, assume the ranking of EECS 660 is: Mary (1st), Tom (2nd), and Jacob (3rd)
  - and the ranking of EECS 560 is: Tom, Jacob, and Mary.
  - we can associate the ranking with the students in EECS 660, and map the rankings to EECS 560
    - Mary is 1, Tom is 2, and Jacob is 3; In EECS 560, Tom (2) goes first, so we have 2 as the first mapped ranking; Jacob (3) goes second, so we have 3 as the second,… eventually we get the mapped ranking as 2, 3, 1

COUNTING INVERSIONS

- Then, we define an inversion in a list A as the number of cases where $a_i > a_j$ if $i < j$. Here, a_i is the ith element of A.
  - for example, consider the mapped ranking in EECS 560, which is 2, 3, 1
  - for any combination of two elements, i.e., $\{(2,3), (2,1), (3,1)\}$, we know that $(2,1)$ and $(3,1)$ are inversions per the definition
  - thus, the number of inversions we have in EECS 560's mapped ranking is 2
  - if two lists are identical, then we have 0 inversion; if two lists are completely reversed, then for any pair of elements it is an inversion ($\binom{n}{2}$ inversions in total)
  - **in this case, we can use the number of inversions to quantify the similarity of two rankings**

COUNTING INVERSIONS

- Counting inversions: Given an unsorted list of n numbers $1,2, \ldots, n$, where each number occurs exactly once, count the number of inversions in the list.

- Naïve solution: we simply compare each pair of numbers, and we get a time complexity of $O(n^2)$.

COUNTING INVERSIONS

- To solve the problem using divide and conquer, note that we can divide the entire set of numbers into two subsets, and the total number of inversions we have in the entire list is the sum of the number of inversions in the first list, the number of inversions in the second list, and the number of inversions between the two lists.
  - using the naïve $O(n^2)$ algorithm, we know that each of these three numbers can be computed in $T(n)/4$, and the sum of them is $3T(n)/4$, so we should be getting some efficiency gain here (very conceptual idea, NOT a valid proof)
  - however, per the Master's theorem, time complexity for the merge phase is in fact a lower found for the entire algorithm; we have that complexity being $O(n^2)$, hence the naïve divide and conquer algorithm has no gain in theoretical efficiency

COUNTING INVERSIONS

- Hence, to improve the time complexity, we shall do better in merging the results of two subproblems, i.e., counting the number of inversions between two subsets.

- Key observation: while we cannot count the number of inversions between two subsets with an efficiency better than $O(n^2)$, we can actually count it easily if the two subsets are sorted.

COUNTING INVERSIONS

- Let $A_1 = \{2,6,1\}$ and $A_2 = \{3,5,4\}$, we can sort them individually into $A'_1 = \{1,2,6\}$ and $A'_2 = \{3,4,5\}$.
- Per definition, we know that all numbers in A'_2 should be larger than those in A'_1, if there is no inversion (all numbers in A_2 should have larger indexes than those numbers in A_1).
- So, for any number in A_2, the number of inversions associated with it is exactly the number of numbers in A_1 that are larger than it.
- Since A_1 is sorted, obtaining such a number should be easy.

COUNTING INVERSIONS

- Consider $A'_1 = \{1,2,6\}$ and $A'_2 = \{3,4,5\}$.
  - 3 is smaller than 6, count 1 inversion
  - 4 is smaller than 6, count 1 inversion
  - 5 is smaller than 6, count 1 inversion
  - 3 between-subset inversions in total (consider originally $A_1 = \{2,6,1\}$ and $A_2 = \{3,5,4\}$)

COUNTING INVERSIONS

- Consider $A'_1 = \{1,2,6\}$ and $A'_2 = \{3,4,5\}$.
- We can implement the idea in a way similar to merge sort:
  - we set up three pointers (for input arrays A'_1, A'_2 and an output array A')
  - when we need to move any number in A'_2 into A', it indicates that the number is smaller than all of the remaining numbers in A'_1, hence it will cause that many inversions
  - for example, we know at the point of $A'_1 = \{6\}$ and $A'_2 = \{3,4,5\}$, we shall be moving the number 3 in A'_2 to A'; there is 1 number remaining in A'_1, so there is 1 between-subset inversion associated with 3.
  - this process clearly takes $O(1)$ time

COUNTING INVERSIONS

- A happy coincidence: the process of counting between-subset inversions also sorts the merged set! (And our assumption that the subsets are sorted can thus be satisfied!) This process is thus called merge-and-count.

- The overall running time thus becomes:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

- And per the Master's theorem, the time complexity is $O(n \log n)$.

COUNTING INVERSIONS

```
Merge-and-Count(A,B)
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  Maintain a variable Count for the number of inversions,
    initialized to 0
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    If b_j is the smaller element then
      Increment Count by the number of elements remaining in A
    Endif
    Advance the Current pointer in the list from which the
      smaller element was selected.
  EndWhile
  Once one list is empty, append the remainder of the other list
      to the output
  Return Count and the merged list
```

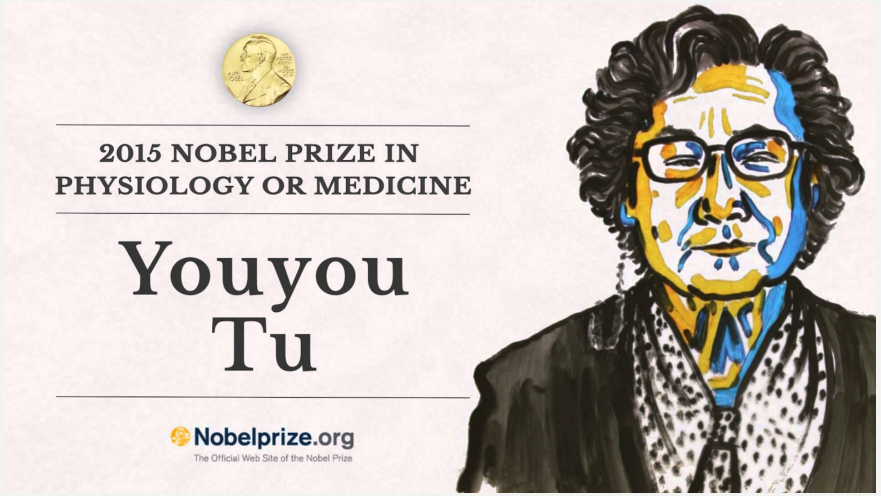
CLOSEST PAIR OF POINTS

- The closest pair of points problem, as the name suggests, tries to find the closest pair of points (in terms of Euclidean distance) among $n \geq 2$ points in a d dimensional space.

- The problem has a lot of applications in computational geometry, and we will try to motivate the problem using a case in computational drug design.

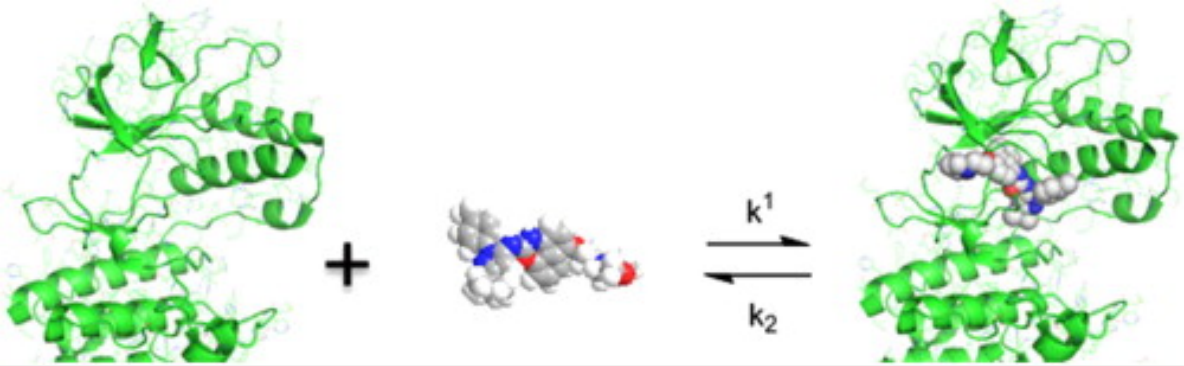
CLOSEST PAIR OF POINTS

- Drug design is the design of a molecule that can specifically bind to the expected bucket of the target protein.
  - Many drugs were discovered from nature
  - For example, many drugs were extracted from traditional Chinese herbal medicine: Dr. Youyou Tu discovered the anti-malaria drug artemisinin from *Artemisia Annua* (or Sweet Wormwood), and won the Nobel Prize in Physiology and Medicine in 2015.

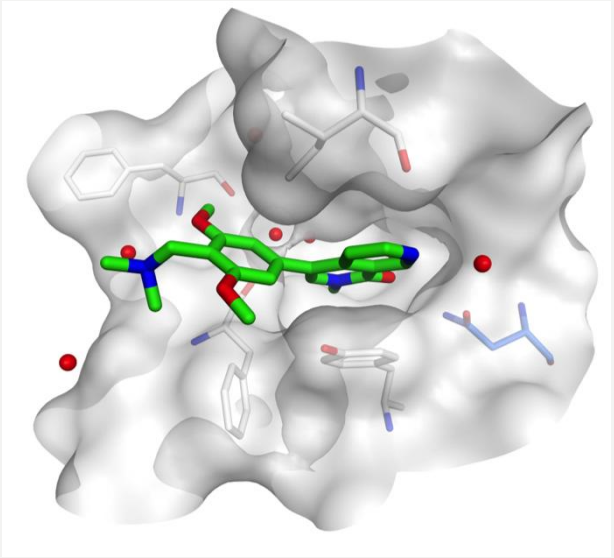


CLOSEST PAIR OF POINTS

- In many cases, we have to design our own molecular structures (and synthesize them from scratch) to cure specific diseases. There are two fundamental requirements for drug design:
  - specificity: the drug should not bind with non-targeted proteins; since we do not want the drug to "kill" normal proteins and cause side effects
  - effectiveness: the drug should bind tightly and to the expected bucket of the target protein; as we want the treatment to be effective and long-lasting



https://ars.els-cdn.com/content/image/1-s2.0-S0960894X15001341-fx1.jpg



https://www.drugdiscoverytrends.com/pinpointing-how-new-drugs-exert-beneficial-effects/

CLOSEST PAIR OF POINTS

- The naïve way (unfortunately, the most popular way implemented by the big pharms nowadays) is to exhaustively dock the drug candidate with all proteins
  - we can move and rotate the drug molecules, and identify the way that it best fits a protein
  - then, we will calculate the distance between the **contacting atoms** of the drug and the protein, to see if they are within the expected distance
  - while we can visually see which atoms are contacting from the 3D image, computers only see two sets of coordinates (a set for the drug and another for the protein)
  - in this case, identifying the **closest pair of between-molecule points** as the **anchor** provides us an effective mean to quickly locate the binding site, and we can then calculate the contact distances of their surrounding atoms

CLOSEST PAIR OF POINTS

- However, drug binding modeling is a tedious task:
  - there are many drug candidates
  - there are many proteins in human (other biomolecules such as DNAs and RNAs too)
  - a drug candidates can be moved and rotated in many different ways in the 3D space
  - we see that the number of closest pair of points instances we need to solve is the product of the above three factors

CLOSEST PAIR OF POINTS

- So, what is the naïve way to solve the problem?

- We can simply compute the Euclidean distance between any two points, leading to an $O(n^2)$ time complexity.
  - this could be infeasible for real-world applications
  - interestingly, we note that in the 1-dimensional case (where the points are distributed in a single line), we can solve this problem in $O(n \log n)$ time
  - we simply sort the points according to their 1-dimensional coordinates and scan the sorted points to find the closest pair

CLOSEST PAIR OF POINTS

- Unfortunately, with higher dimensions, we cannot use the same strategy.
  - consider 2D cases
  - points that are closer in one dimension does not mean they are also close in both dimensions (say (1,1) and (1,100) vs (2,1) and (1,2))

- But the 1-dimensional case suggests that it could be possible to reduce the time complexity from $O(n^2)$ to $O(n \log n)$ with a careful algorithmic design.

CLOSEST PAIR OF POINTS

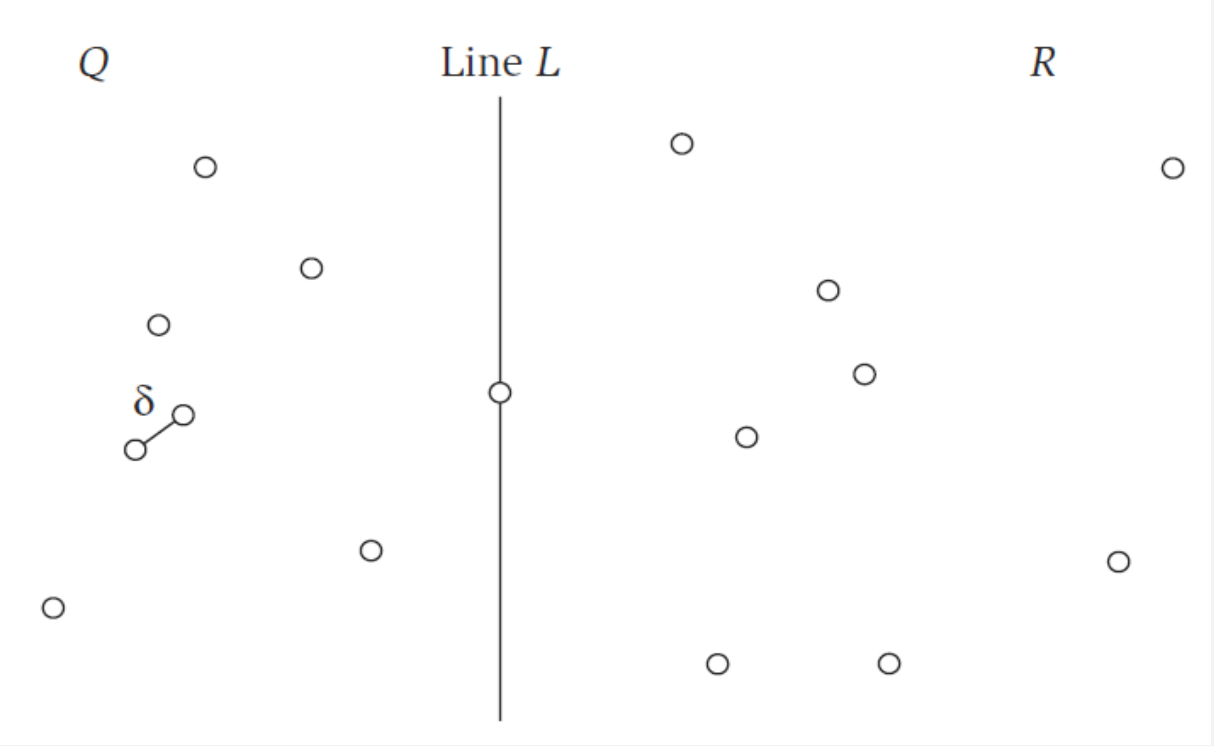
- Using the idea similar to counting inversions, we can split the entire set of points into two equal-sized subsets.
  - to find the closest pair of points within each subset, we would need $T(n)/4$ time (assuming the naïve algorithm with $O(n^2)$ time complexity)
  - to find the closest pair of points between subsets, we would also need $T(n)/4$ time
  - so, we can solve the problem in $3T(n)/4$ time
  - unfortunately, we still have the overall $O(n^2)$ time complexity, unless we are able to find between-subset closest pair of points in $O(n)$ time (i.e., a linear time merge for the two subsets)

CLOSEST PAIR OF POINTS

- Now assume that we are working on a 2D version of the problem, i.e., each point is associated with a coordinate (x, y).

- First, let's figure out how to divide the set of points:
  - since the points scatter on a 2D plane, we can simply cut the space into two subspaces such that each subspace contains half of the points
  - to prevent complicated algebraic computation, we will simply cut the space based on the x dimension
  - that is, we simply cut the plane with a vertical line $x = x^*$, where half of the points have their x coordinates less than x^*
  - for simplicity, denote the line $x = x^*$ as L, and the two subset of points as Q and R

CLOSEST PAIR OF POINTS



- We will also denote the closest pair of points within Q or R as δ, which can be found by recursively solving the problem on each subset.

CLOSEST PAIR OF POINTS

- Conceptually, δ gives us an upper bound for between-subset distance
  - if the between-subset distance is larger than δ, we may simply return δ as the closest distance of the entire set
  - it indicates that our possible search space should be a band containing L as its middle axle and with a width of δ

CLOSEST PAIR OF POINTS

Proposition: If there exists $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$, then each of q and r lies within a distance δ of L (the line used for separating Q and R).

Proof: Denote the x coordinate of L as x^*. If q and r exist, denote their coordinates as $q = (x_q, y_q)$ and $r = (x_r, y_r)$. By definition of the division of subproblems, we have $x_q \leq x^* \leq x_r$. Then, we have

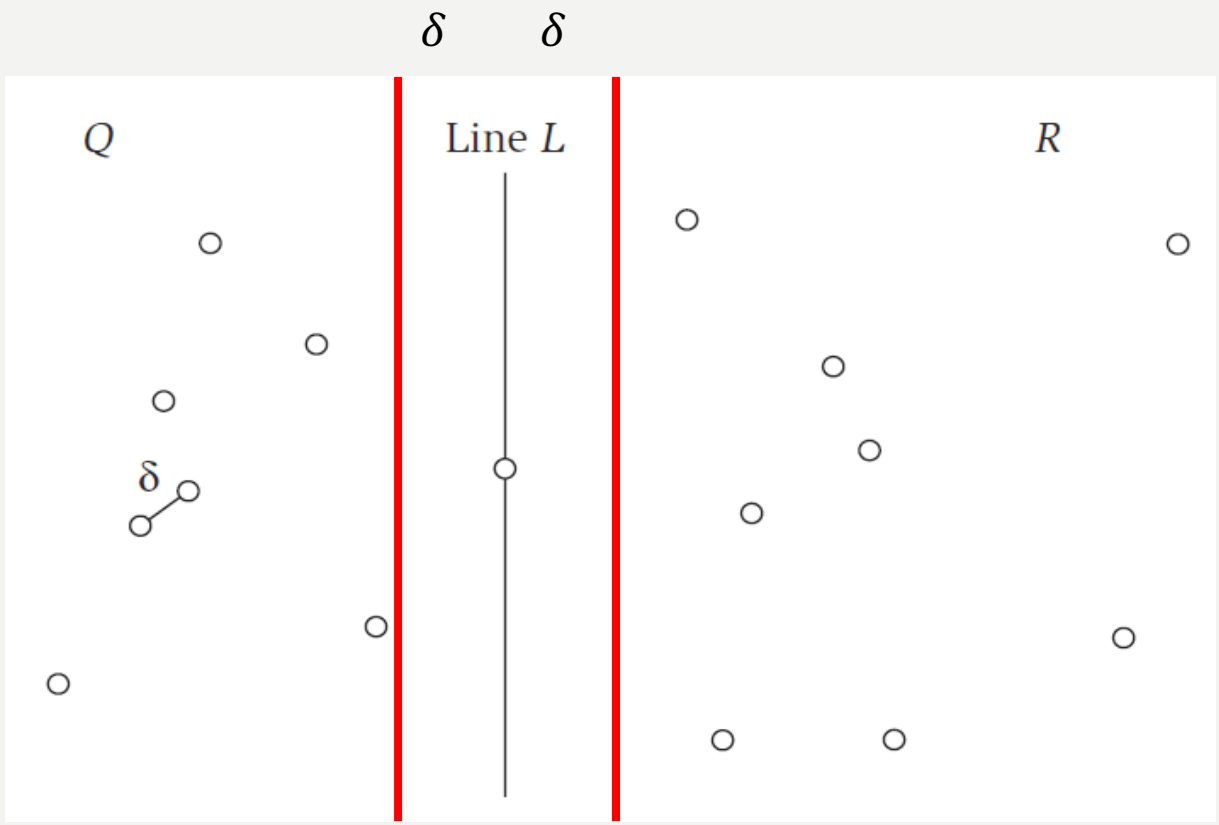
$$x^* - x_q \leq x_r - x_q \leq d(q,r) = \delta$$

and

$$x_r - x^* \leq x_r - x_q \leq d(q,r) = \delta$$

So, each of q and r has an x coordinate within δ of x^* and hence lies within distance δ of L.

CLOSEST PAIR OF POINTS



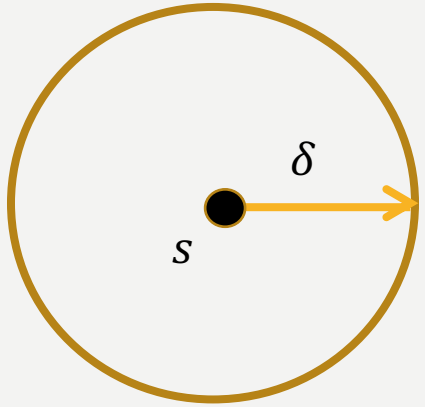
CLOSEST PAIR OF POINTS

- Now, let all points within the band be S.

- Clearly we only need to check all pairs of points in S, which is smaller than all points.
  - however, it could be possible that all points are aggregated within the band
  - so, in the worst-case scenario we still need $O(n^2)$ to compute the distance between all pairs of points
  - we need to do even better

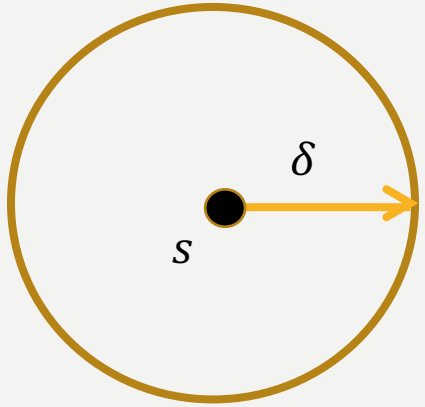
CLOSEST PAIR OF POINTS

- Given a point s within S, what we are interested in is to see if there exists some other points within a circle induced by p with a radius of δ.
  - we are not interested in points outside the circle because the corresponding distance will be larger than δ
  - that is, we can further restrict the set of points that could lead to a smaller distance than δ with s



CLOSEST PAIR OF POINTS

- If we analyze further, we will realize that the points that could lead to a distance smaller than δ with s must satisfy the following two conditions:
  - the points must not be within the same subset as s (in other words, these points must all be in the opposite subset of s)
  - the points must be within the circle



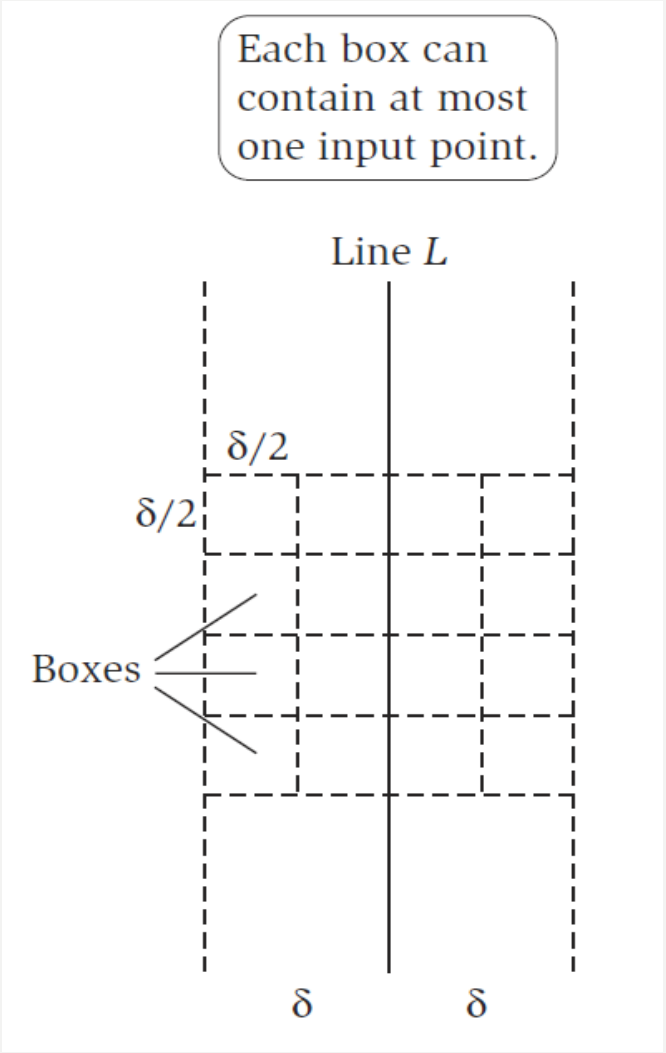
CLOSEST PAIR OF POINTS

- We further know that, within the circle, we can only hold a limited number of points within the same subset.
  - this is because for any two points within the same subset, their distance must be larger than δ (recall that δ is the smallest within-subset distance)
  - if a point exists within the circle, it excludes the possibility that another point within its surrounding area with a radius of δ
  - and we claim that the maximum number of points that can exist within the circle is upper bounded by a constant

- To help with ordering the points, we assume that we have sorted all the points in S according to their y coordinates, and denote the corresponding sorted list as S_y.

CLOSEST PAIR OF POINTS

- **Proposition**: If $s, s' \in S$ have the property of $d(s, s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S_y.
- **Proof**: We will grid the band into small boxes with sizes of $\frac{\delta}{2} * \frac{\delta}{2}$. Note that no two points can be within the same box; because the farthest distance for two points within the box is $\sqrt{2}\delta/2$ (the diagonal), contradicting with the minimum within-subset distance of δ. So, in the worst case, we have one point in each small box. Clearly, if two points differ from each other more than 15 positions, their distances will be at least $\frac{3\delta}{2} > \delta$.



CLOSEST PAIR OF POINTS

- As a result, for any point $s \in S_y$, we only need to compute the distance between s and its next 15 points.
  - from the 16th point and beyond, it is impossible that they will be within a distance of δ with s
- Now, we have reduced the time complexity for the distance calculation to $O(n)$
  - but we still need to sort the y coordinates to form S_y
  - a trick to obtain the sorted set is: we pre-sort all points in their y coordinates (for only once before the algorithm begins), and form S_y by checking the points one-by-one to see whether their x coordinates are within the band. If yes, we insert that point into S_y.

CLOSEST PAIR OF POINTS

- Now, we can perform all merge operations in $O(n)$ time
  - $O(n)$ construction of S_y
  - $O(n)$ time scan of each point s in S_y and distance calculation with its next 15 points

- Taken together, the divide and conquer algorithm has:
  - 2 subproblems ($a = 2$)
  - each subproblem has half of its original size ($b = 2$)
  - linear time merging ($c = 1$)
  - using the Master's theorem, the time complexity of the algorithm is $O(n \log n)$

CLOSEST PAIR OF POINTS

```
Closest-Pair(P)
  Construct P_x and P_y  (O(n log n) time)
  (p_0^*, p_1^*) = Closest-Pair-Rec(P_x, P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
    find closest pair by measuring all pairwise distances
  Endif

  Construct Q_x, Q_y, R_x, R_y (O(n) time)
  (q_0^*, q_1^*) = Closest-Pair-Rec(Q_x, Q_y)
  (r_0^*, r_1^*) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))
  x^* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x^*}
  S = points in P within distance δ of L.

  δ = min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))
  x^* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x^*}
  S = points in P within distance δ of L.

  Construct S_y (O(n) time)
  For each point s ∈ S_y, compute distance from s
     to each of next 15 points in S_y
     Let s, s' be pair achieving minimum of these distances
     (O(n) time)

  If d(s,s') < δ then
     Return (s,s')
  Else if d(q_0^*, q_1^*) < d(r_0^*, r_1^*) then
     Return (q_0^*, q_1^*)
  Else
     Return (r_0^*, r_1^*)
  Endif
```

SUMMARY

- Three additional divide and conquer algorithms
  - integer multiplication
  - counting inversions
  - closest pair of points