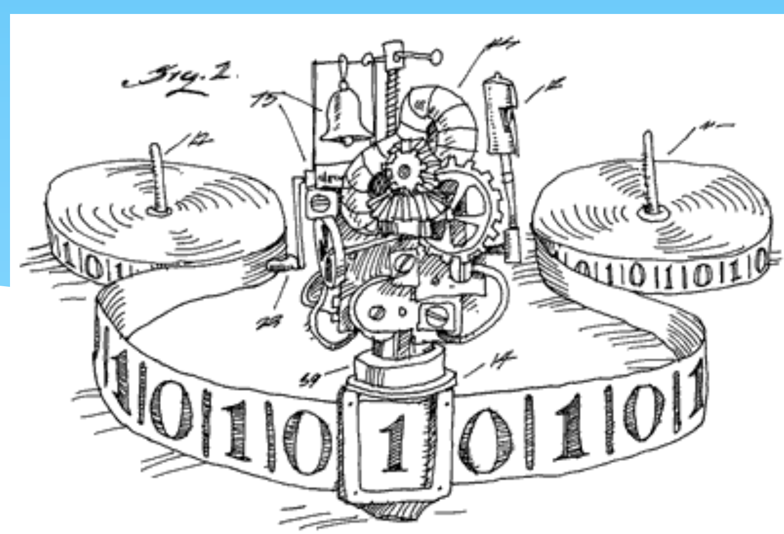
EECS 376: Foundations of Computer Science

Seth Pettie

Lecture 2



Today's Agenda

* Review of the Euclidean Algorithm
  * Using Potential Function
* Divide and Conquer algorithms
  * Mergesort
  * Closest pair

Greatest Common Divisor

* **Definition:** Let $x, y \in \mathbb{N}$ (natural numbers). The Greatest Common Divisor (gcd) of x and y is the largest $z \in \mathbb{N}$ that divides x and y.
  * If gcd(x,y) = 1 then x and y are called **coprime**.

* **Examples:**
  * $gcd(21,9) = 3$
  * $gcd(121,5) = 1$ (co-prime)
  * $gcd(81,48) = \text{gcd}(3^4, 2^4 \cdot 3) = 3$

* **Naïve Algorithm:**
  * For z from y down to 1:
    * If $((z|y) \wedge (z|x))$, return z.

Runtime: $\boldsymbol{O(y)}$ operations.
If x, y are n-bit numbers, $O(y) = O(2^n)$ is exponential in the input size!

We want an algorithm that is polynomial in the input size, e.g., $O(n), O(n^2), etc.$

EECS M ECE CSE

Step 3: Think about the “structure” of the problem.

- **Strategy:** Recursively solve the problem, by reducing to *smaller* numbers.
- Suppose $x \geq y$. Observe: $\gcd(x, y) = \gcd(y, x - y)$.
- Proof: If d divides both x and y, d also divides $x - y$.

  Conversely, any d that divides both $x - y$ and y also divides x.

  So the common divisors of x, y are the common divisors of $y, x - y$

  Hence, their **greatest** common divisors are equal.

- In general, we can reduce k times until $x - ky < y$.
- **Q:** What is $x - ky$?
  - $x \bmod y$ = the remainder of x divided by y.

- **Thereom:** $\gcd(x, y) = \gcd(y, x \bmod y)$

A good potential function

* The ***sum*** of the arguments to **Euclid** decreases quite rapidly.

Euclid(x, y): // for $x \geq y > 0$
  if($x \bmod y = 0$), return y.
  else return **Euclid**$(y, x \bmod y)$

* Define x_t, y_t to be the arguments to the tth call to **Euclid**, where $x_t \geq y_t$.
* Define the ***potential*** to be $s_t = x_t + y_t$.

* **Claim.** $s_{t+1} < \frac{2}{3} s_t$.

A good potential function

Euclid(x, y): // for $x \geq y > 0$
if$(x \; mod \; y \; = \; 0)$, return y.
else return **Euclid**$(y, x \; mod \; y)$

* **Claim.** $s_{t+1} < \frac{2}{3} s_t$.
* **Proof.** Write $x_t = k_t y_t + r_t$, where $k_t \geq 1, r_t < y_t$
  * What is $x_{t+1} = ?$ $\boxed{y_t}$
  * What is $y_{t+1} = ?$ $\boxed{r_t}$

* $s_t = x_t + y_t = k_t y_t + r_t + y_t \geq 2y_t + r_t$
* $> 2y_t + r_t - \frac{y_t - r_t}{2} = \frac{3}{2}(y_t + r_t) = \frac{3}{2} s_{t+1}$.

A good potential function

Euclid(x, y): // for $x \geq y > 0$
if($x \bmod y = 0$), return y.
else return **Euclid**$(y, x \bmod y)$

* **Claim.** $s_{t+1} < \frac{2}{3} s_t$.

* Thus, if there are t calls to **Euclid,** $2 \leq s_t < \left(\frac{2}{3}\right)^t (x+y)$
* Which implies that $t < \log_{3/2}((x+y)/2)$
* $n = \log_2 x + \log_2 y$ is the ***input size*** (bits).
* $t < \log_{3/2}((x+y)/2) < \log_{3/2} x + \log_{3/2} y = \left(\log_{3/2} 2\right) n$.

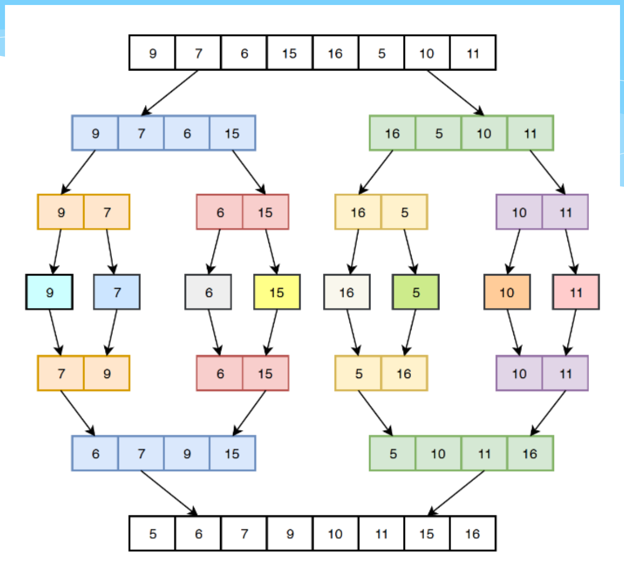
The number of recursive calls is ***linear*** in the ***number of digits*** in the input.

The optimal analysis

* Is there a better bound than $\log_{3/2}(x+y)$?
* The actual maximum recursion depth is $\log_{\phi}(x+y)$, where $\phi = \frac{\sqrt{5}+1}{2} \approx 1.618$ is the ***golden ratio***.
  * Can you prove this?

“Divide et impera” – Philip II

Algorithmic Strategy:
Divide and Conquer



Template

- If the input is a "*base case*" of the problem:
  - *directly* compute the answer and return it
- Otherwise:
  - *divide* the problem into *smaller* subproblems
  - *recursively* solve each subproblem
  - *combine* the solutions

Example: MergeSort

MergeSort(A[1..n]): // sorts a list of integers
if n = 1 then return A // base case
L = **MergeSort**(A[1..n/2]) // recursively sort 1^{st} half
R = **MergeSort**(A[n/2+1..n]) // recursively sort 2^{nd} half
return **Merge**(L, R) // combine solutions

6	**14**	**12**	**1**	**9**	**4**	**8**	**0**	**5**	**13**	**15**	**10**	**7**	**2**	**3**	**11**

Sort each half recursively

0	**1**	**4**	**6**	**8**	**9**	**12**	**14**	**2**	**3**	**5**	**7**	**10**	**11**	**13**	**15**

Merge

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

EECS ECE CSE

Combining two *sorted* lists

MergeSort(A[1..n]): // sorts a list of integers
if n = 1 then return A // base case
L = **MergeSort**(A[1..n/2]) // recursively sort 1^{st} half
R = **MergeSort**(A[n/2+1..n]) // recursively sort 2^{nd} half
return **Merge**(L, R) // combine solutions

* The heart of the **MergeSort** procedure is how we **Merge** the two *sorted* sublists, L and R
* **Idea:** repeatedly compare the front of L and R; pop off the smaller one and append it to the merged list

L

3	6

R

2	4	5

2	3	4	5	6

Analysis of MergeSort

MergeSort(A[1..n]): // sorts a list of integers
if n = 1 then return A // base case
L = **MergeSort**(A[1..n/2]) // recursively sort 1st half
R = **MergeSort**(A[n/2+1..n]) // recursively sort 2nd half
return **Merge**(L, R) // combine solutions

* We expect you to analyze the *runtime* (and sometimes *correctness*) of *each* algorithm that you design

* **Runtime:** for $n \geq 1$, let $T(n)$ be the runtime of **MergeSort** on a list of n integers. We can write $T(n)$ as a *recursive function* (***recurrence***):

$$\underbrace{T(n)}_{\text{time to MS a list of } n \text{ integers}} = \begin{cases} O(1) & n = 1 \\ \underbrace{T\left(\frac{n}{2}\right)}_{\text{time to MS 1st half}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{time to MS 2nd half}} + \underbrace{O(n)}_{\text{time to Merge}} & n > 1 \end{cases}$$

Note: we typically omit the base case

Analysis of MergeSort

```
MergeSort(A[1..n]): // sorts a list of integers
if n = 1 then return A              // base case
L = MergeSort(A[1..n/2])            // recursively sort 1st half
R = MergeSort(A[n/2+1..n])          // recursively sort 2nd half
return Merge(L, R)                  // combine solutions
```

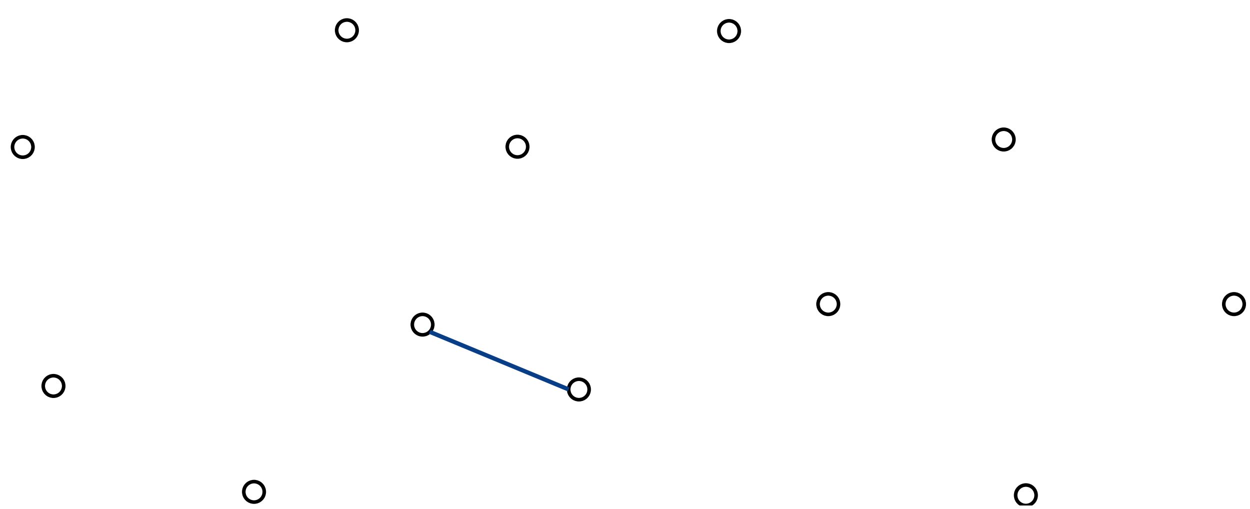
* **Runtime:** for $n \geq 1$, let $T(n)$ be the runtime of **MergeSort** on a list of n integers. We can write $T(n) = 2T(n/2) + O(n)$.

  (**Next time:** tool to show $T(n) = O(n \log n)$.)

* **Correctness:** Strong induction on size of list, n.
  * As a base case, **MS** is correct on lists of size 1. Now suppose **MS** is correct on lists of size $< n$. Then **MS** is correct on 1st /2nd half, by assumption. Since **Merge** is correct, **MS** is correct on n.

Example: Closest Pair in 2D

* Given a set of $n \geq 2$ points in the *plane*.
* **Goal:** Find ***minimum distance*** between any pair of points.
* A point $p = (x_p, y_p)$ is represented by a pair of numbers.
* (Pythagorean Theorem) $dist(p,q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$.
* **How fast is the trivial algorithm for this problem?**



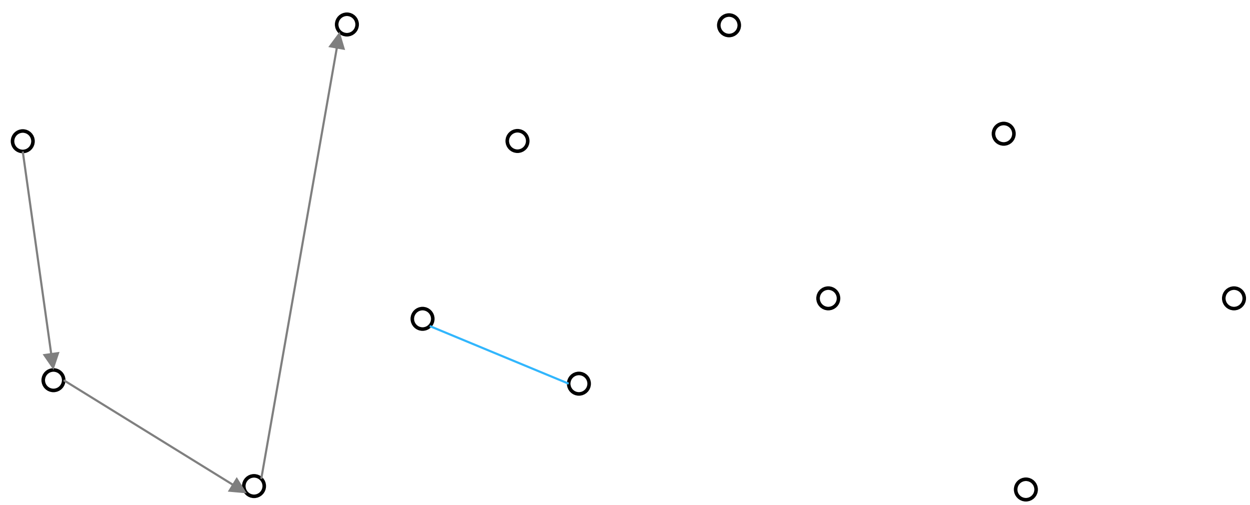
Example: Closest Pair in 1D

* You're given a set of $n \geq 2$ *distinct* points on a line.
* **Goal:** Find minimum distance between any pair of points
* **Q:** Can you think of a fast algorithm?
  * (1) Sort the points in increasing order as $(p_1, p_2, \ldots, p_n)$ — $O(n \log n)$ time
  * (2) Scan the list of sorted points; return $\min_{1 \leq i < n}\{p_{i+1} - p_i\}$. — $O(n)$ time

Example: Closest Pair in 2D

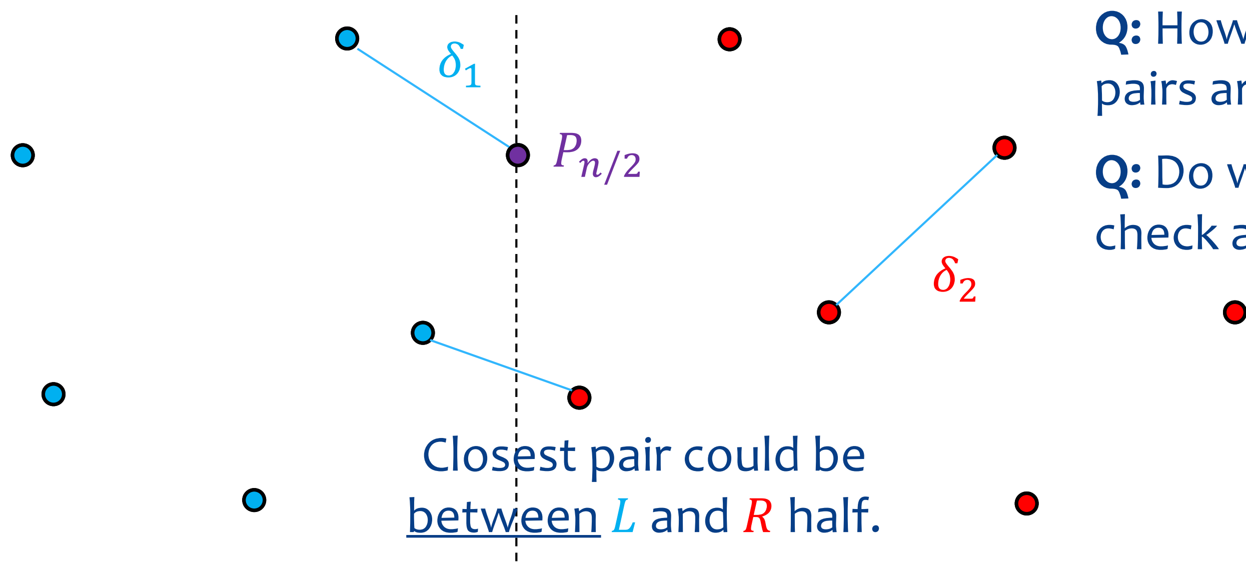
- Given a set of $n \geq 2$ *distinct* points in the *plane*.
- **Goal:** Find minimum distance between any pair of points

- **Q:** What goes wrong with "walk left to right" strategy?
  - Might need to check all previous points; $O(n^2)$ runtime



Divide and Conquer?

ClosestPair$(P_1, \dots, P_n)$: // $n \geq 2$ pts in the plane, x-sorted asc.
if $n \leq 3$ then return min dist among P_1, P_2, P_3 // base case
$(L, R) \leftarrow$ partition points by $P_{n/2}$ // split by median in x-coord
$\delta_1 \leftarrow$ **ClosestPair**(L) // min dist on left n/2 pts
$\delta_2 \leftarrow$ **ClosestPair**(R) // min dist on right n/2 pts
… What comes next? // *… how?*
… Just return $min\{\delta_1, \delta_2\}$*?*



Closest pair could be <u>between</u> L and R half.

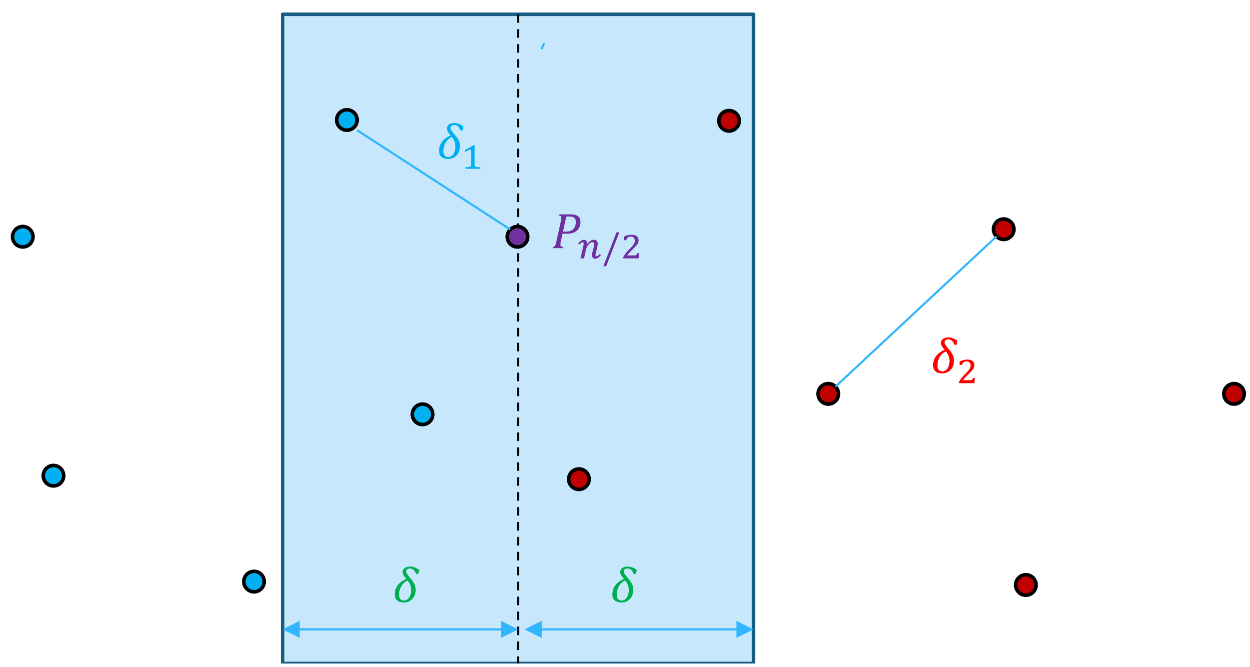
Q: How many blue/red pairs are there?

Q: Do we need to check all of them?

The δ-strip

ClosestPair$(P_1, \dots, P_n)$: // $n \geq 2$ pts in the plane, x-sorted asc.
if $n \leq 3$ then return min dist among P_1, P_2, P_3 // base case
$(L, R) \leftarrow$ partition points by $P_{n/2}$ // split by median
$\delta_1 \leftarrow$ **ClosestPair**(L) // min dist on left
$\delta_2 \leftarrow$ **ClosestPair**(R) // min dist on right
need to know min dist between L and R // … look at δ-strip

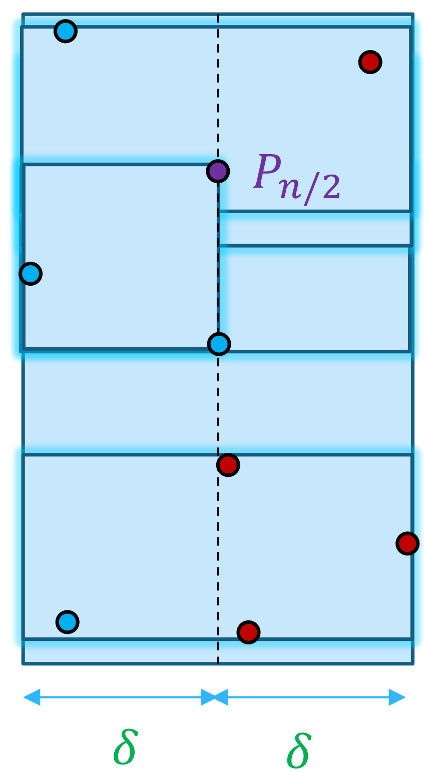
* Let $\delta = \min\{\delta_1, \delta_2\}$.
* **Observation:** We can focus on points whose x-coord is within δ of $P_{n/2}$'s x-coord (the "δ-strip").



Properties of the δ-strip

ClosestPair$(P_1, \ldots, P_n)$: // $n \geq 2$ pts in the plane, x-sorted asc.
if $n \leq 3$ then return min dist among P_1, P_2, P_3 // base case
$(L, R) \leftarrow$ partition points by $P_{n/2}$ // split by median
$\delta_1 \leftarrow$ **ClosestPair**(L) // min dist on left
$\delta_2 \leftarrow$ **ClosestPair**(R) // min dist on right
need to know min dist between L and R // …look at δ-strip

- Let $\delta = \min\{\delta_1, \delta_2\}$.
- **Q:** How many pts can there be in the δ-strip?
- **Q:** How many blue pts can there be in a $\delta \times \delta$ square?
- **Q:** How many pts can there be in a $\delta \times 2\delta$ rectangle?



How to find a close red/blue pair:
Slide a $\delta \times 2\delta$ rectangle!

Analysis of ClosestPair

ClosestPair$(P_1, \ldots, P_n)$: // $n \geq 2$ pts in the plane, x-sorted asc.
if n = 2 then return dist(P_1, P_2) // base case
$(L, R) \leftarrow$ partition points by $P_{n/2}$ // split by median
$\delta_1 \leftarrow$ **ClosestPair**(L) // min dist on left
$\delta_2 \leftarrow$ **ClosestPair**(R) // min dist on right
Let $(P'_1, P'_2, \ldots, P'_m)$ be points in the δ-strip, //m≤n
sorted by *y-coordinate*
$\delta_3 \leftarrow \min_{1 \leq i < m, 1 \leq c \leq 7}\{dist(P'_i, P'_{i+c})\}$ // ≤7m distances computed
return $\min\{\delta_1, \delta_2, \delta_3\}$

* **Runtime:** For $n \geq 2$, let $T(n)$ be the runtime of **ClosestPair** on n points.
  * $T(n) = 2T(n/2) + O(n \log n)$
  * How can we improve this to $T(n) = 2T(n/2) + O(n)$?

Aside: A lower bound on sorting

- **Fact:** If the numbers can only be *compared* (e.g., $A[i] < A[j]$?), then *any* sorting algorithm requires at least $\log_2(n!) = \Theta(n \log n)$ *comparisons* to sort a list of n distinct numbers.
- **Idea:** The algorithm must be able to distinguish the *total* order of the elements, e.g., $A[3] < A[1] < A[2]$.
  - We can use a potential function argument to show that it can't do this too quickly (we'll play the role of an *adversarial input*)

Aside: A lower bound on sorting

The interaction might look something like this:

Alg. Query	Our Ans.
$A[1] < A[2]$?	Yes
$A[2] < A[3]$?	No

In general: Always choose the answer that results in *more* indistinguishable orderings.

Indistinguishable orderings

1,2,3	2,1,3
1,3,2	2,3,1
3,1,2	3,2,1

Aside: A lower bound on sorting

Define s_t = ***number of possible orderings*** after t comparisons.

Define potential $\Phi_t = \log_2(s_t)$.

There are ***2 possible answers*** to the $(t+1)^{\text{th}}$ comparison. If we pick the one that max. s_{t+1}, then $s_{t+1} \geq \frac{1}{2} \cdot s_t$ and $\Phi_{t+1} \geq \Phi_t - 1$.

We're done sorting when $s_t = 1$ and $\Phi_t = 0$,

So $t \geq \Phi_0 = \log_2(n!) = n \log n - O(n)$.