TAXONOMY OF ALGORITHMS

- Serial vs Parallel
  - Serial: One instruction at a time
  - Parallel: Multiple instructions at once

- Deterministic vs Randomized
  - D: On multiple runs on same input, always do same thing
  - R: On multiple runs on same input, may do different things
    Example: flip a coin, and base your next action on the result

- Exact vs Approximate
  - Exact: exact solution to the problem
  - Approximate: produce something “close” to a solution

Why deterministic?
- we want absolute guarantees
- we want to learn about analysis
  (and this is harder for randomized)

Implications:
- do not use hashing
- do not use maps
**word-RAM model:** for input of n integers, assume a word is large enough to address n locations, thus each word has log n bits. So the input integers can have \( O(\log n) \) bits, but bigger integers must be split into multiple words.

**bit model:** count bits and bit operations.

**Examples:**
- a. given a permutation of 1..n (e.g. to sort)
- b. given a number (e.g. to test if it's prime)
- c. given a list of n natural numbers \( s_1, s_2, \ldots s_n \) (e.g. to sort)
Which model should we use and what is the input size?

Examples:
- a. given a permutation of 1..n (e.g. to sort) – or n numbers in range 1..c.n
- b. given a number (e.g. to test if it’s prime)
- c. given a list of n natural numbers $s_1, s_2, \ldots, s_n$ (e.g. to sort)

(a) use word RAM – each input number fits in one word.
   - Sorting $O(n \log n)$
   - Bit model: input size $O(n \log n)$
   - Sorting $O(n \log n)$ comparisons $\times O(\log n)$ per comparison

(b) use bit model.
   - or use word RAM – allows only constant word size
     - so same as bit model.

(c) use word RAM but pay attention to sizes of numbers.
Running Time of a Program: $T_M(I)$ denotes the running time (in seconds) of a program $M$ on a problem instance $I$.

Worst-case Running Time as a Function of Input Size: $T_M(n)$ denotes the \textit{maximum} running time of program $M$ on instances of size $n$:

$$T_M(n) = \max\{T_M(I) : \text{Size}(I) = n\}.$$  

Average-case Running Time as a Function of Input Size: $T_M^{\text{avg}}(n)$ denotes the \textit{average} running time of program $M$ over all instances of size $n$:

$$T_M^{\text{avg}}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{I : \text{Size}(I) = n} T_M(I).$$

Why worst case? We want an \textbf{absolute guarantee}.

Why $O(\cdot)$ (or $\Theta(\cdot)$)? To \textbf{ignore which machine} we pay attention to the big picture.
Tricky Question

2 Algs. with run times:

\[ O(n^2) \quad O(n \log n) \]

which is better? - We can't say.

Could be \( \Theta(n) \)

If \( \Theta \) - then \( \Theta(n \log n) \) better than \( \Theta(n^2) \)
big O

\[ f(n) \in O(g(n)) \text{ is like } \leq \]
\[ f(n) \in o(g(n)) \text{ is like } < \]
\[ f(n) \in \Theta(g(n)) \text{ is like } = \]

For numbers, we have a total order.

For any \( x, y \) either \( x < y \), \( x = y \), \( x > y \)

Is big O a total order?

No. Use \( \sin \) to build oscillating function.
big O

\[ f(n) \in O(g(n)) \text{ is like } \leq \]
\[ f(n) \in o(g(n)) \text{ is like } < \]
\[ f(n) \in \Theta(g(n)) \text{ is like } = \]

For numbers, we have a total order.

Is big O a total order? No.

Challenge Exercise: find functions f, g, such that neither is big O of the other, but use only +, x, log, exponentiation. Don’t use ceiling, floor, sin, etc.
Analyzing a program with branching

```
max := 0;
for i := 1 to n do
    for j := i to n do
        sum := 0;
        for k := i to j do
            sum := sum + A[k];
        if sum > max then max := sum;
```

$\Theta(n^3)$
Question 1. Suppose you want to store an array and Add elements to it. When the array is full, create a new array of twice the size and copy the current array into it.

Suppose the array has \( n \) elements in it and we call Add. How much time will Add take in the worst case? \( \Theta(n) \) if we need to double

Suppose we start with an array of size 1 and Add \( n \) elements. How much time does this take in the worst case?

\[
\sum_{j=1}^{\log n} 2^j = O\left(2^{\log n + 1}\right) = O(n).
\]
Reductions

**Question 2.** Given $n$ points in the plane, $p_i = (x_i, y_i)$ sort by $x$ then by $y$.

$\begin{align*}
\text{Y} = 3 & \quad \text{ } \\
\text{Y} = 2 & \quad \text{ } \\
\text{Y} = 1 & \quad \text{ } \\
\text{Y} = 0 & \quad \text{order} \\
\end{align*}$

$x = 1 \\
x = 2 \\
x = 3$

Suppose you have a program, Sort, that sorts a list of $n$ numbers in $\Theta(n \log n)$. How can you sort $n$ points in the plane in $\Theta(n \log n)$?

1. suppose you can modify/generalize the Sort program
2. suppose you cannot. Give a reduction — you may call Sort more than once
3. Give a many-one (“one-shot”) reduction — you may only call Sort once.

1. replace $<$ for numbers by comparing points
   
   if $x_1 < x_2$ then $p_1 < p_2$
   
   if $x_1 > x_2$ then $p_1 > p_2$
   
   if $x_1 = x_2$

2. Need **stable** Sort. Sort by $y$, then by $x$.

3. challenge. convert $(x_i, y_i)$ to a single number.