Introduction

Randomized Algorithms: Week 1
Summer HSSP 2023
Emily Liu

About this class

- **Survey** class of randomized algorithms
  - Each week is a different algorithm
- Alternative view of programming/algorithms
- Prerequisites:
  - Some programming experience
    - Level: AP Computer Science, USACO Bronze, or equivalent
  - Comfortable with probability concepts
    - We’ll do some review!

About me

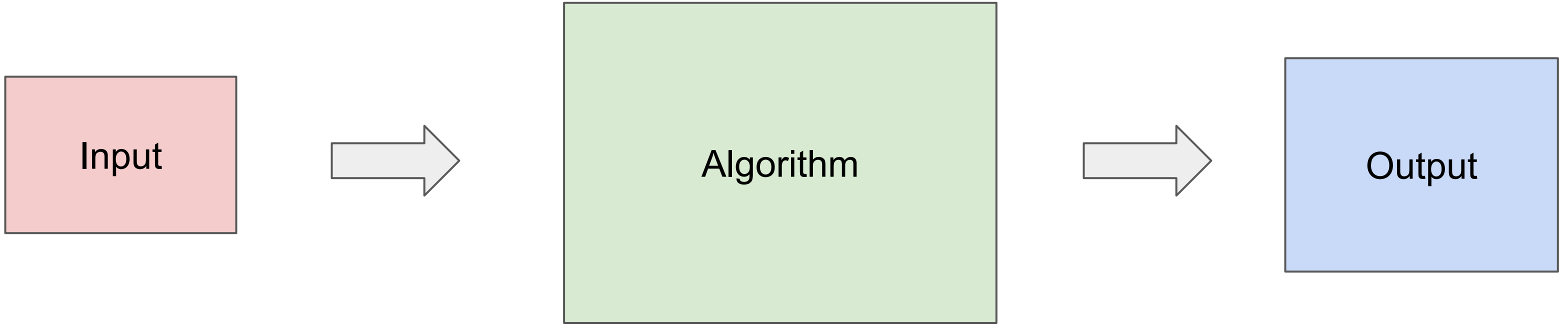
- Emily Liu
- From California (Bay Area!)
- Undergrad at MIT (co2024), doubling Computer Science + Math
- Currently: software internship, machine learning research

About **YOU**

- Grade
- Where are you dialing in from?
- First time at HSSP?
- Preferred programming language OR best thing you ate this week

What is a randomized algorithm?

Deterministic Algorithms

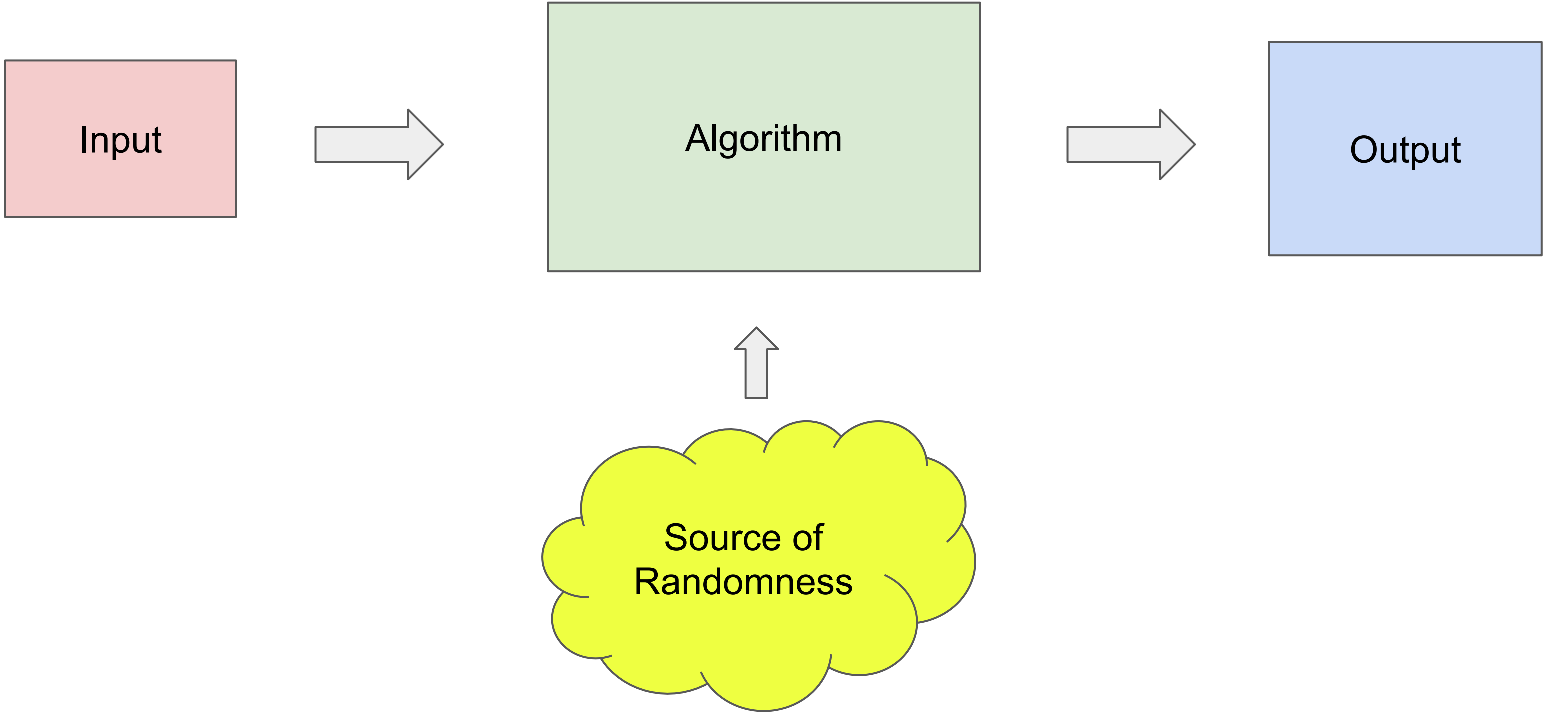


Input

Algorithm

Output

Randomized Algorithms



Question: Why randomization?

1. Deterministic algorithms can be too slow
2. Randomized algorithms are *good enough*

Two classes of randomized algorithms:

Monte Carlo

Guaranteed runtime,
Probabilistic correctness

Las Vegas

Probabilistic runtime,
Guaranteed correctness

Monte Carlo Algorithms

- Will always run in **polynomial time**

  i.e: O(n^k) where k is a constant
- The probability of being wrong is upper-bounded by some small value **ε**
- Biases:
  - **False-biased:** Always correct when returning false, sometimes correct when returning true
  - **True-biased:** Always correct when returning true, sometimes correct when returning false
  - **Unbiased:** Symmetric probability of success for true or false return

Las Vegas Algorithms

- **Always** returns the correct output
- The algorithm itself may run for longer or shorter depending on
  1. The input
  2. The random numbers
- We look at the **expected value** of the runtime.
  - Formally, this should still be polynomial time.

Some silly examples

Task: Given an array A of length N and a target value k, determine if the value k exists in A.

```
contains(A, k):
    i = randInt(0 … N-1)
    return (A[i] == k)
```

Questions:

- Is this a Las Vegas or Monte Carlo algorithm?
- Is this a false-biased, true-biased, or unbiased algorithm? Why?
- How can we be more confident in our answer, without writing any new code?

Some more silly examples

Task: Given an array A of length N with exactly one value equal to a target value k, return the index i such that A[i] = k.

```
find_index(A, k):
    i = randInt(0 … N-1)
    return i if (A[i] == k) else find_index(A, k)
```

Questions:

- Is this a Las Vegas or Monte Carlo algorithm?
- Is this algorithm guaranteed to terminate?
- How many times in expectation will the algorithm repeat?

Mathematical building blocks

Random variables, probability distributions

Random variable: variable that can take on multiple values, determined by a random function

Are described through **probability mass functions** (discrete) or probability density functions (continuous)

Examples

- Outcome of a coin toss or dice roll
- Output of Monte Carlo Algorithms
- Runtime of Las Vegas Algorithms

Uniform random variables

Discrete uniform distribution

X takes on n possible values, all with equal likelihood (1/n).

```
contains(A, k):
    i = randInt(0 … N-1)
    return (A[i] == k)
```

randInt is a discrete uniform random variable that takes on values from 0 through N-1.

Geometric random variables

Models a process where you repeat something independently, and terminate at each step with a probability p.

Example: Toss a fair coin until it lands on heads.

```
find_index(A, k):
    i = randInt(0 … N-1)
    return i if (A[i] == k) else find_index(A, k)
```

The number of times find_index has to run is a geometric random variable. What is p?

Expected value

Given a random variable X,

E[X] = sum(p(x) * x) over all values x that X can take on.

Linearity of expectation: E[aX + bY] = a E[X] + b E[Y]

Exercises:

1. Calculate expected value of a uniform random variable that takes on values from 1 to N, where N is a positive integer.
2. Calculate expected value of a geometric random variable with a probability parameter p.

Revisiting contains

If A does not contain k, algorithm is always correct.

Assume A contains one copy of k.
We run the function t times.

```
contains(A, k):
    i = randInt(0 … N-1)
    return (A[i] == k)
```

P(incorrect | A contains k) = (1 - 1/n)^t

Probability of failure gets smaller as t increases - but how much?

Revisiting contains

Recall: deterministic algorithm takes n computations

Let t = n: the probability of getting wrong is (1-1/n)^n

-> e^{-1} as n goes to infinity

approx. 0.3, not very good…

Reason: Probability of success (1/n) is very low.

Suppose there were c occurrences of k in A; probability of failure goes to e^{-c}

- Increases as c increases, which makes sense!

Preview of course

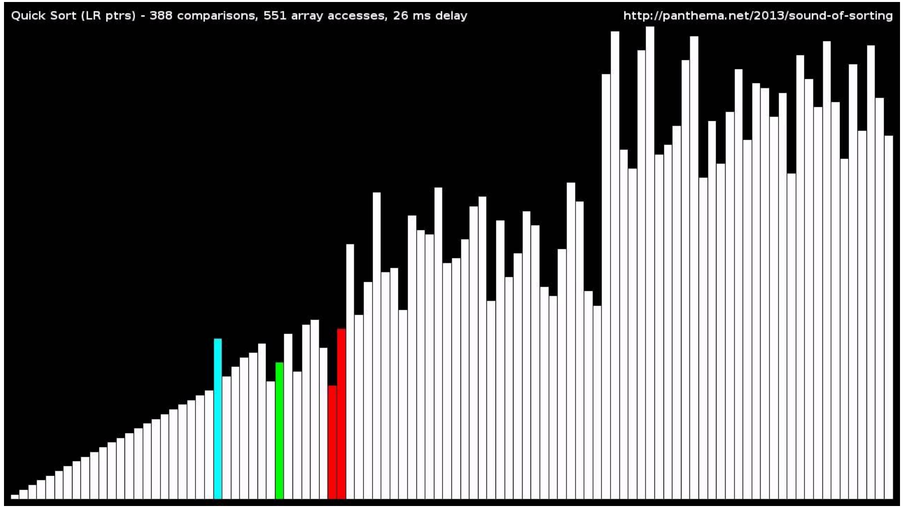
Week 2: Randomized Quicksort

Task: Given a list of comparable values, sort them in ascending order.

Many sorting algorithms already exist

- Selection sort (O(n^2))
- Insertion sort (O(n^2))
- Merge sort (O(n log n))

Quick sort - O(n log n) algorithm with some randomness



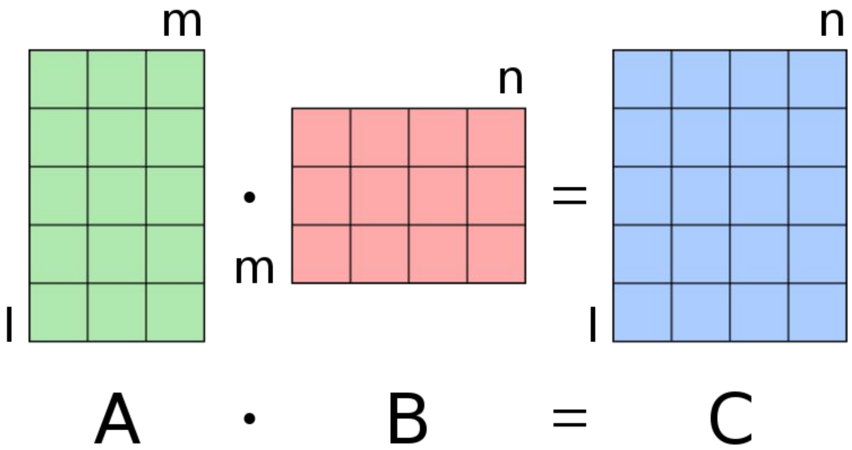
Week 3: Matrix Multiplication

Task: Given three matrices A, B, and C, determine whether or not AB = C.

Traditional matrix multiplication: O(n^3).

Random: Frievalds' algorithm:

O(n^2).



Week 4: Game Tree Evaluation

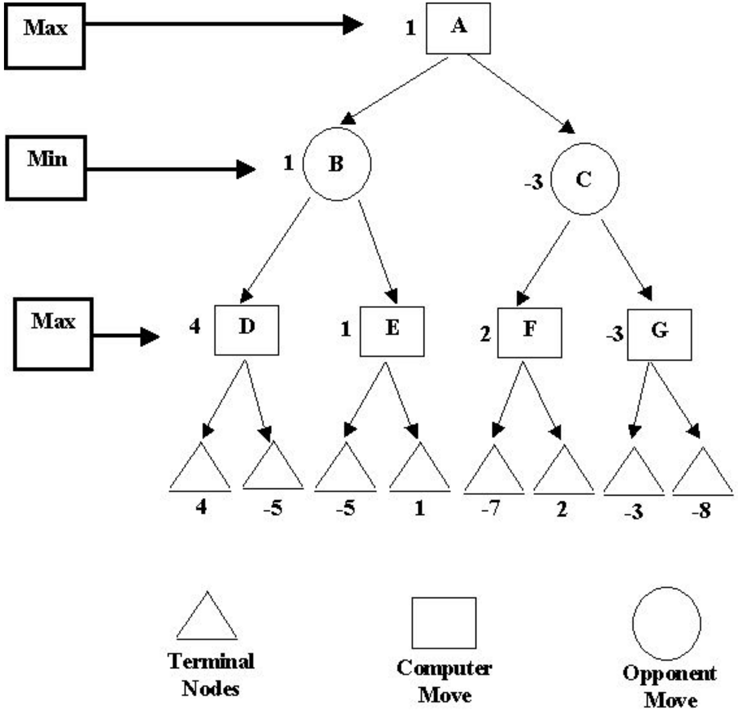
Game tree = rooted tree where every internal node is a MIN or MAX operator

MIN: even distance from root

MAX: odd distance from root

Leaves are numerical values, and at each internal node we apply the operation to all incoming values.

Goal: determine the value at the root.



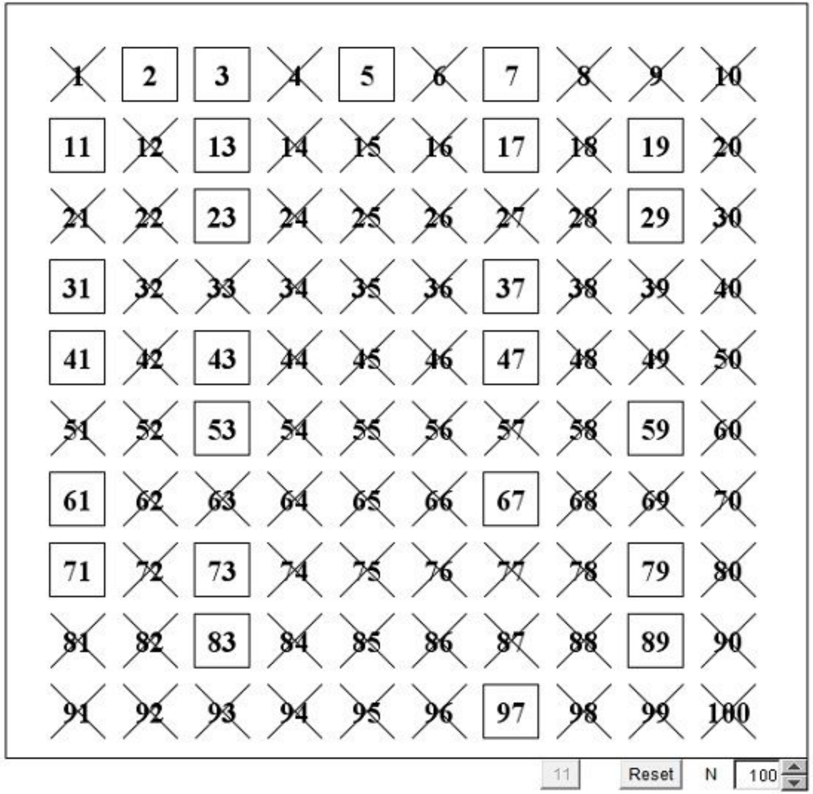
Week 5: Primality Testing

Task: Given a positive integer N, determine whether or not N is prime.

Brute force: O(sqrt(N)) - test every number up to round(sqrt(N)).

Improvement: Memoize prime numbers, test only the primes up to round(sqrt(N))

Randomization can get us faster algorithms!



~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

Week 6: Boolean Satisfiability

Task: Given a boolean expression, determine whether or not there exists an assignment of the variables that makes the expression True

$$(x \text{ OR } y \text{ OR } z) \text{ AND } (x \text{ OR } \bar{y} \text{ OR } z) \text{ AND }$$

$$(x \text{ OR } y \text{ OR } \bar{z}) \text{ AND } (x \text{ OR } \bar{y} \text{ OR } \bar{z}) \text{ AND }$$

$$(\bar{x} \text{ OR } y \text{ OR } z) \text{ AND } (\bar{x} \text{ OR } \bar{y} \text{ OR } \bar{z})$$

Brute force solution: Given n variables, O(2^n).

- Exponential runtime :(
- NP complete: don't really have a better (deterministic) solution

Solution: randomization!