Randomness and Computation

There are many good reasons to let our algorithms make random decisions

- sample a large population / dataset
- avoid pathological worst-case instances
- avoid predictable outcomes
- allow for simpler algorithms

Today we'll cover a few applications of the use of randomness in Algorithms and Data Structures

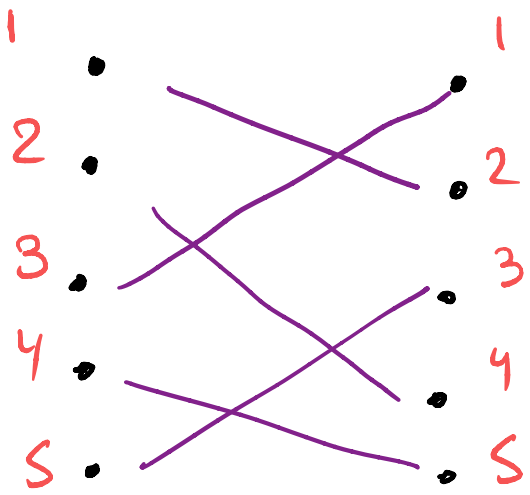
Generating Random Permutations

Input: integer n

Output: permutation of $\{1, 2, \ldots, n\}$ chosen uniformly at random (UAR)

Ex: $n = 5$

$\sigma = \langle 2, 4, 1, 5, 3 \rangle$



Can also think of as shuffling array $\langle 1, 2, \ldots, n \rangle$

Why should we care?

- Many times an algorithm whose input is an array performs better on a randomly shuffled version (Ex, Quicksort)
- Can be used to sample k elements when k is not known in advance
- Can be used to assign scare resources to a set of agents with preferences

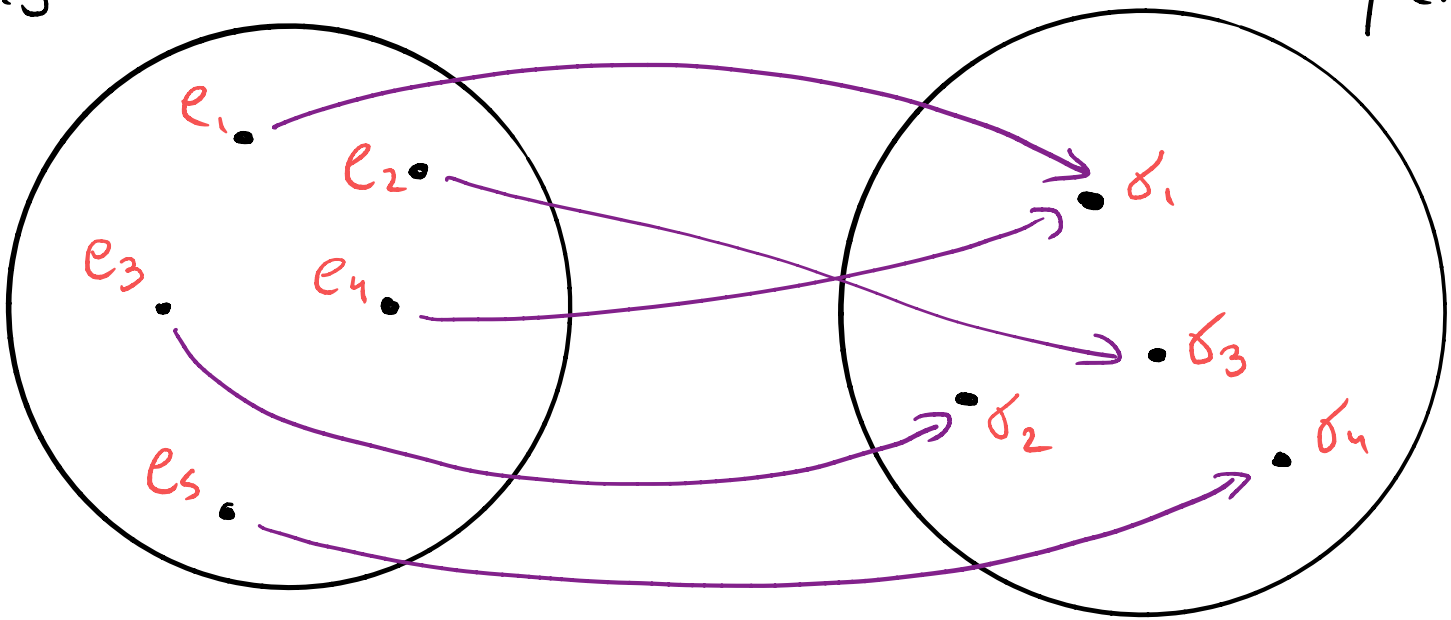
First (incorrect) try

Different executions lead to different outcomes

```
def shuffle(A):
    # permute array A in place
    n ← len(A)
    for i in {0,..., n-1}:
        # swap A[i] with random position
        j ← pick UAR from {0, .., n-1}
        A[i], A[j] ← A[j], A[i]
```

executions

permutations



If every execution is equally likely, then we want that any two permutations are generated by the same # of executions

# of executions $= \underbrace{n \times n \times \cdots \times n}_{n \text{ times}} = n^n$

# of permutations $= 1 \times 2 \times \cdots \times n = n!$

n^n need not divide evenly into $n!$

Ex: $n = 3$

$n^n = 27$

$n! = 6$

and 27 is not a multiple of

First (incorrect) try

```
def fisher_yates_shuffle(A):
    # permute array A in place
    n ← len(A)
    for i in {0,..., n-1}:
        # swap A[i] with random position
        j ← pick UAR from {i, .., n-1}
        A[i], A[j] ← A[j], A[i]
```

# of executions = $1 \times 2 \times \cdots \times n = n!$

# of permutations = $1 \times 2 \times \cdots \times n = n!$

Obs: Each execution leads to different outcome

Ex. from $\langle 1,2,3,4,5 \rangle$ to $\langle 4,2,1,5,3 \rangle$

$\langle 1,2,3,4,5 \rangle$	$i = 0$	$j = 3$
$\langle 4,2,3,1,5 \rangle$	$i = 1$	$j = 1$
$\langle 4,2,3,1,5 \rangle$	$i = 2$	$j = 3$
$\langle 4,2,1,3,5 \rangle$	$i = 3$	$j = 4$
$\langle 4,2,1,5,3 \rangle$	$i = 4$	$j = 4$

Thm: The Fisher-Yates shuffle applies to A a permutation chosen UAR

Proof

Every execution of the algorithm happens with probability $\frac{1}{n!}$

Each execution leads to a different outcome

$\Rightarrow$ $\Pr[\text{FY applies } \sigma] = 1/n!$

for any permutation σ of $\{1, \ldots, n\}$ ∎

Finding Prime Numbers

Def: An integer $p \geq 2$ is a prime if its only divisors are 1 and itself

Finding large primes is an important privitive use in most modern public cyptography systems

"large" means that n has some prescribed number of bit, e.g, 1000 bits

Distribution of Primes

Thm: Let $\pi(n)$ be the number of primes that are $\leq n$, then

$$\pi(n) = \Theta\left(\frac{n}{\ln n}\right)$$

$\Rightarrow$ An iterger n chosen UAR from $\{1, \ldots, N\}$ has a $\Theta\left(\frac{1}{\ln N}\right)$ chance of being prime

$\Rightarrow$ If we can test primality, we are done!

```
def find_prime(N):
    # return a random prime in {1, ..., N}
    do
        n ← pick UAR from {1, ..., N}
    repeat until is_prime(n)
    return n
```

Obs: Let $T(n)$ be running time of is_prime then the expected running time of find_prime is $O(T(N) \log N)$

Testing primality

Rabin-Miller is a randomized algorithm for testing primality that has bounded error

Given n and k,

- if n is prime, $RM(n,k)$ always returns True
- if n is composite, $RM(n,k)$ returns

$$\begin{cases} \text{True} & \text{with prob } \frac{1}{4^k} \\ \text{False} & \text{o.w.} \end{cases}$$

```
def witness(x, n):
    # try to check if n is composite

    write n-1 as 2^k m for m odd
    y ← x^m mod n
    if y mod n = 1 then
        return True    # n is probably prime
    for i in {1, .., k-1} do
        if y mod n = n-1 then
            return True    # n is probably prime
        y ← y^2 mod n
    return False    # n is definitely composite
```

Fact: If $n>2$ is composite there are $\leq \frac{n-1}{4}$ values of x such that $\text{witness}(x, n) = \text{True}$.

If $n>2$ is prime then for all values of x we get $\text{witness}(x, n) = \text{True}$

$\Rightarrow$ If we pick x UAR then

$$\Pr[\text{witness}(x,n) = \text{True} \mid n \text{ is prime}] = 1$$

$$\Pr[\text{witness}(x,n) = \text{True} \mid n \text{ is composite}] \leq \frac{1}{4}$$

Boosting Success Probability

Suppose we call witness(x, n) with k different values of x all chosen UAR, call them $x_1, x_2, \ldots, x_k$

$$\Pr[\text{witness}(x_i, n) = \text{True} \ \forall i \mid n \text{ is composite}]$$

$$= \prod_{i=1}^{k} \Pr[\text{witness}(x_i, n) = \text{True} \mid n \text{ is composite}]$$

$$\leq \prod_{i=1}^{k} \frac{1}{4} = \frac{1}{4^k}$$

$\Rightarrow$ We can boost success probability with k

```
def is_prime(n, k):
    # check if n is prime
    # may return True if n is composite w.p. 1/4^k

    repeat k times
        x ← pick UAR from {2, .., n-1}
        if not witness(x, n)
            return False

    return True
```

Thm: If n is prime, `is_prime(n, k)` always returns True

If n is composite, `is_prime(n,k)` returns

$$\begin{cases} \text{True} & \text{with probability } \leq 1/4^k \\ \text{False} & \text{otherwise} \end{cases}$$

The algorithm performs $O(k \log n)$ arithmetic operations

Treap

In Assignment 5 given $\{(v_i, p_i)\}_{i=1}^{n}$ you had to build a binary tree that was at once:

- BST with respect to v_i
- have heap property with respect to p_i

Advanced student had to design an algorithm for inserting a new item (v, p) in $O(h)$ time, where h is tree height

Treap

Thm: If p_i is chosen UAR from $[0,1]$ then for a Treap on $\{(v_i, p_i)\}_{i=1}^{n}$ we have

$$E[\text{Treap height}] = \Theta(\log n)$$

Therefore we can get a balanced BST with a very simple data structure

---

```
def insert_balanced_BST(v):
    p ← pick UAR from [0,1]
    insert_treap(v, k)
```

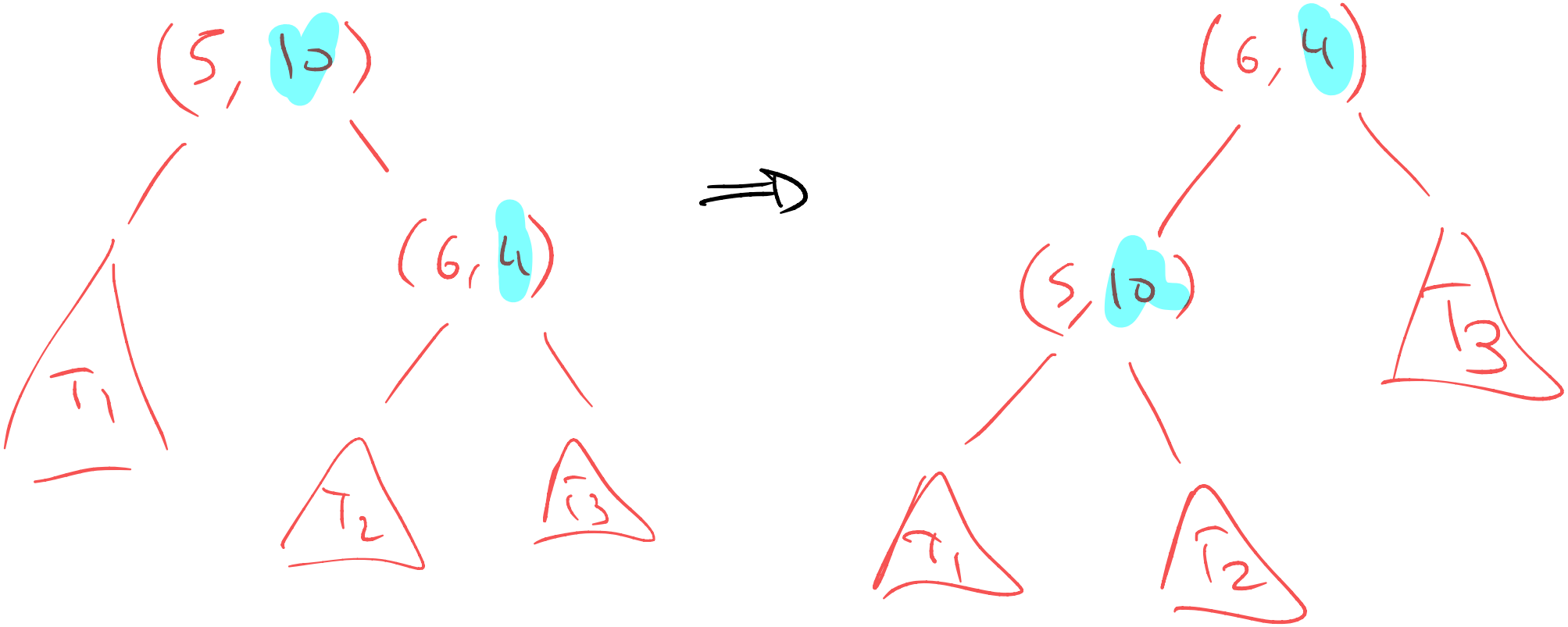
Random priority key is chosen on the fly

Obs: insert takes expected $O(\log n)$ time

Treap insert

Do a regular BST insertion

Do local rotations to restore heap property

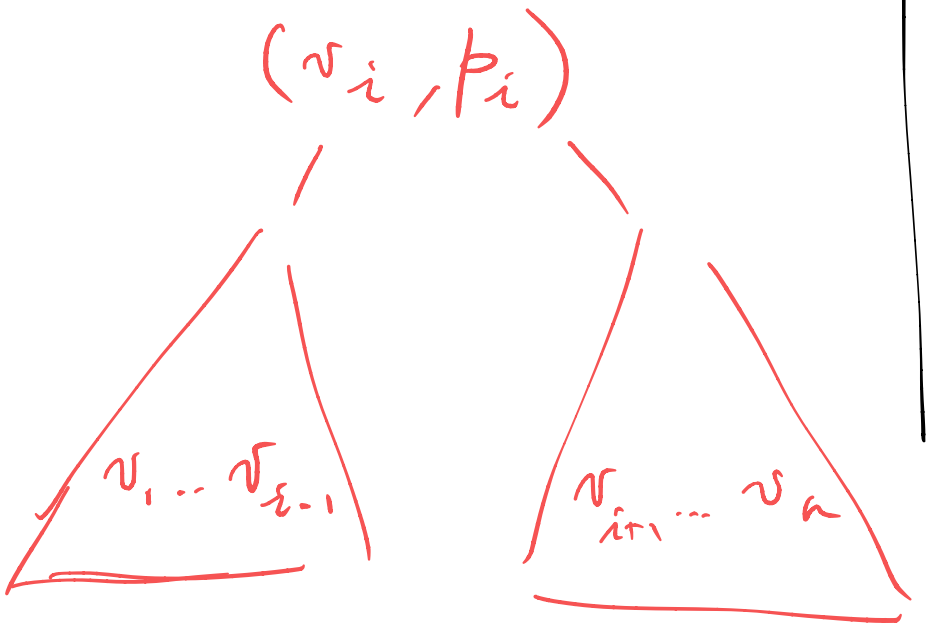


Treap Height

Suppose we sorted values so that $v_1 \leq v_2 \leq \cdots \leq v_n$

$\Pr[v_i \text{ is the root}] = \frac{1}{n}$

$\Pr[\text{root} \in \{v_{\frac{n}{4}}, \ldots v_{\frac{3}{4}n}\}] = \frac{1}{2}$



$\Rightarrow$ Most nodes are fairly balanced

$\Rightarrow$ $O(\log n)$ height with high probability

Admin

- Gradescope (Assignment 9)
- Gradescope (Assignment 10)
  - put your files at root of zip file
  - edit LANGUAGE file
  - don't leave it for the last moment!
- Next week we do a recap and solve a final exam
- Quiz 10 (about the final)