Week12 Randomness

Randomness is basically an ingrediant that we can add to our algorithm.

Usage of Random decisions:

- Sample a large population / dataset — training model (eg. Machine Learning)
- Avoid pathological worst-case instance for the algorithm you design — eg. Quick sort (bad perfromance in pathological input)

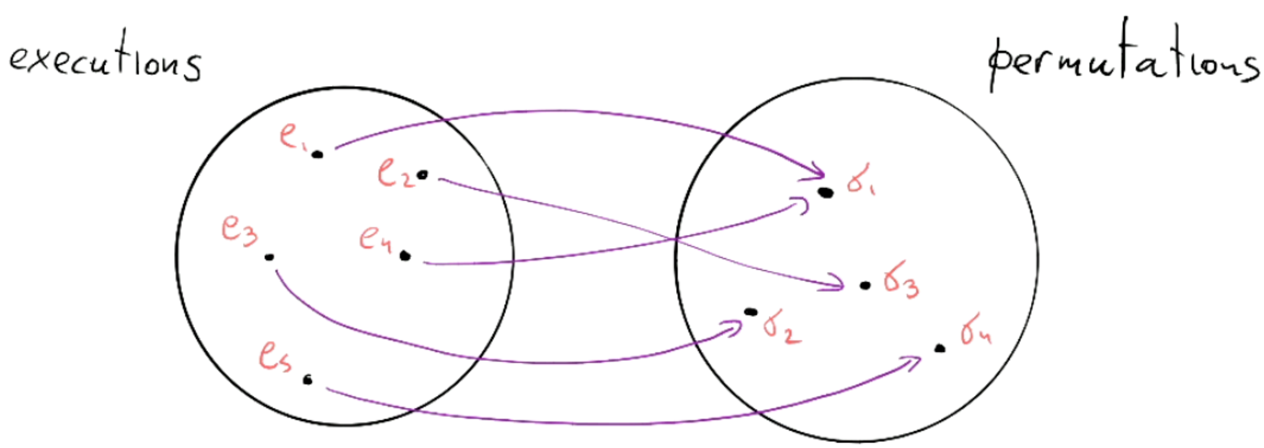
Generating Random Permutation on given input

- can be used in shuffle instance
- sample things from large puplation — pick up small branch from shuffled given dataset

First (Incorrect) sample on lecture slides

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

When we compute, there are several execution of this algorithm that will depend on this random random number j. So different execution will lead to different outcome (we call it different permutation).



Those random decision made by algorithm and assigned to j , it mapes deterministically to a certain output (or permutation). (from e_i to b_i on the graph) And, after that, all steps of the algorithm is deterministic.

In our case, all of those execution are equally likely (each number in {0, ..., n-1} has equally likely to be picked and assigned to j). Because we are saying that we are picking number UAR (uniformally and random). There is no bias when j is picked from {0, ..., n-1}.

Then, what we want is two different permuation will have the same number of executions producing them. Otherwise, it wouldn't be a unifiormed distribution of generating permutation.

Let's see if there are same number of executions are mapped to all different permutations.

> Number of executions:
>
> For each iteration of the for loop we need to pick index j . So for the first iteration, we have n chooses to pick j, second iteration, we also have n chooses to picked j .... etc. For each iteration, we have same amount of choose to pick j.
>
> Therefore, there are n^n executions.

> How many different available permutations we have on each iteration? n, n-1, n-2, ..., 3, 2, 1, 0
>
> Total permutations may generate : $1 * 2 * 3 * ... * n = n!$

If these algorithm was an algorithm produce permutation uniformly and randomly. Then we will need to divide the number of execution evenly into our number of outcomes (permutations).

> n^n % $n!$ is not integer

Corrected version:

```
def shuffle(a):
  # permute array A in place
  n <- len(A)
  for i in {0, ..., n-1} do
    # Swap A[i] with the random position in front of i
    j <- pick uniformly and random (UAR) from {i, ..., n-1}
    a[i], a[j] <- a[j], a[i]
```

The Fisher Yates shuffle

It takes an array A and applies permutation to a choosen uniformally and random. And every execution of this algorithm is equally likely.

Proof:

- Every execution of the algorithm happens with probability $1/n!$.
- Each execution leads to a different outcome.
- => Therefore the Probability of the Fisher Yates applies on a given permutation σ is 1/n!:
  - Pr[$Fy_{applies}\sigma$] = 1/$n!$, and this is true for any permutation σ of {1, ... , n}

Finding Prime Numbers

Def: an integer $p => 2$ is a prime if its only divisors are 1 and itself.

Finding large primes is an important primitive use in most modern public cryptography systems

- "Large" means that n has some prescribed number of bit, e.g. 1000bits
- this is the example how randomness lead to pretty elegant algorithm
- Used in pseudo random generator

Distribution of Primes:

Number of theories proved that prime numbers are plentiful.

Theme: Let $\pi(n)$ be the number of primes that are $\leq n$, then the number of prime number is $\pi(n) = \theta(n/logn)$

- In another word, pick $logn$ numbers from the given set, we can find one prime number in average.
- An integer n choosen UAR from {1,...,n} has a $\theta(1/logn)$ chance of being prime.
- If we can test primality, we are done!

```
def find_prime(n):
  do:
    n <- pick UAR from {1,...,n}
  repeat until is_prime(n)
  return n
```

Let $T(n)$ be running time of is_prime then the expected running time of find_prime is $O(T(n)logn)$

Testing primality

Rabin-miller primality testing algorithm is a randomized algorithm for testing primality has bounded error.

Given n and k,

- if n is prime, RM(n) always returns TRUE
- if n is composite, RM(n) return:
  - TRUE, with $1/4^k$
  - FALSE, otherwise

```
# Robin-miller
def witness(x,n):
  # try to check of 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 definitly composite
```

The property of this algorithm has is that:

- If n > 2 is composite there are $\leq (n-1)/4$ values of x such that $witness(x,n)$ = TRUE
- If n > 2 is prime then for all values of x we get $witness(x,n)$ = TRUE
- If we pick x UAR then:
  - Pr[$witness(x,n) = TRUE$ | n is prime] = 1
  - Pr[$writness(x,n) = TRUE$ | n is composite] $\leq$ 1/4

Treap

In Assignment 5 given $\{(v_i, p_i)\}_{i=1}^n$, you have to build a binary tree taht 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 $O(n)$ time, where n is tree height.

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

$E[Treap\_height] = \theta(logn)$

Therefor 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,p)
```

- Random priory key is chosen on the fly
- Obs: Insert takes expected $O(logn)$ time

Treap height

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

v_i ending up at the root of Treap is only depends on the priority you get. And you need to get the smallest primority for v_i and that is the only way you can let v_i end up at the root. Therefore:

> Pr[v_i get the smalles priority]
>
> = Pr[v_i is the root]
>
> = 1/n

, cause each priority value will be pick equally likely.

When v_i is the root, that implies that from $v_1 \ldots . v_{i-1}$ are on the left subtree of v_i , and else goes to the right. Picking perfect $v_i = n/2$ has really small probability, only 1/n. However, landing in the middle of this range from 1 ... n (the root v_i is in range $v_{n/4}, \ldots, v_{3n/4}$) is 1/2. Therefore, the resulted Treap is sort or balanced. Most nodes are fairlt balanced.

Neither left nor right from root has $3n/4$ of the elements with hight probability.

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