Data Management and Artificial Intelligence

Lecture 12

Sebastian Wandelt (小塞)

Beihang University

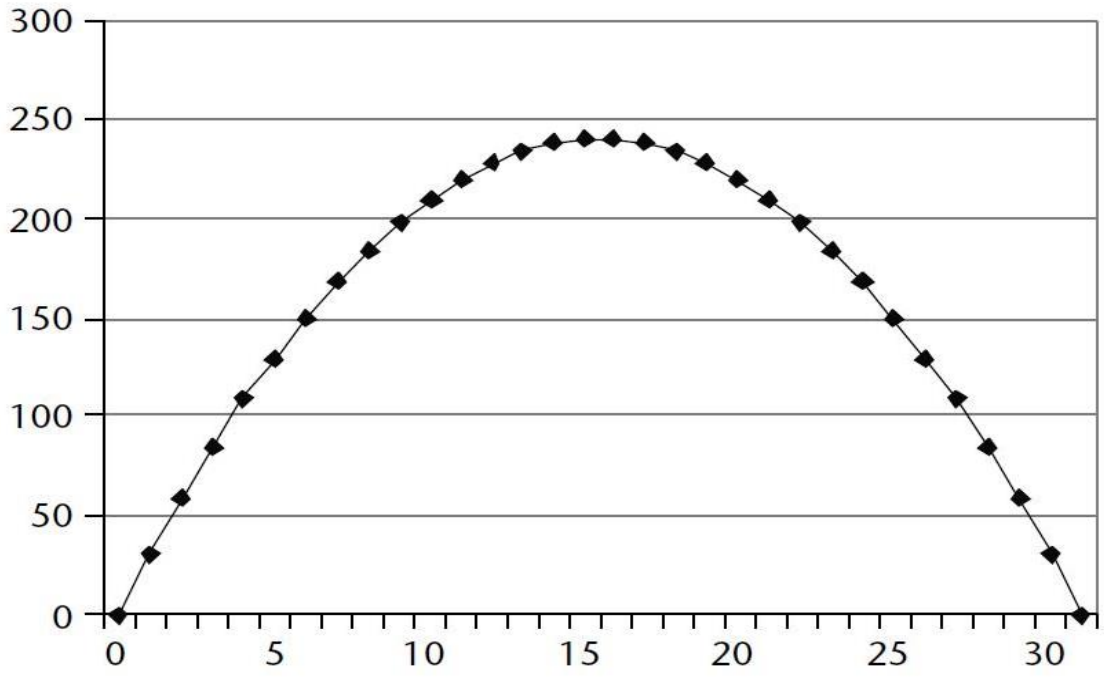
Outline for today

- Review
- Multi-Armed Bandit Problem
- Monte Carlo Methods
- Monte-Carlo Tree Search (MCTS)

Review

Challenge: GA for the following problem

- Find maximum value of a function f(p)=31*p* – *p*² with a single integer parameter *p (0<=p<=31)*



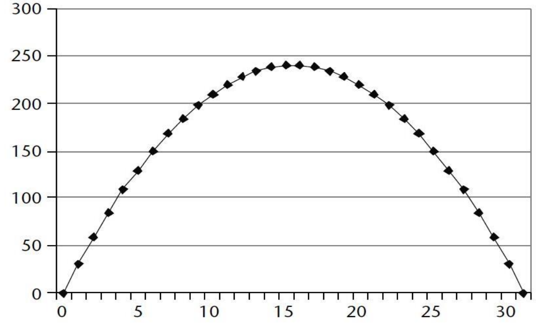
Chromosome: Two options

1. An individual chromosome is a 1-integer number
   - 22=22
2. An individual chromosome is a 5-bit number
   - **10110**=$\mathbf{1}*2^4+\mathbf{0}*2^3+\mathbf{1}*2^2+\mathbf{1}*2^1+\mathbf{0}*2^0$=22

Chromosome example

- An individual chromosome is a 5-bit number
  - **10110**=**1***2^4+**0***2^3+**1***2^2+**1***2^1+**0***2^0=22
- Four randomly generated genomes

Genome	p	Fitness
10110	22	198
00011	3	84
00010	2	58
11001	25	150



Recombination

Genome	p	Fitness
10110	**22**	**198**
00011	3	84
00010	2	58
11001	**25**	**150**

- Recombination of 10110 and 11001 after bit 2:
  - Parent1: 10 110 (22)
  - Parent2: 11 001 (25)
- Offspring (before mutation):
  - Offspring1: **10001** (17) -> fitness=238
  - Offspring2: **11110** (30) -> fitness=30

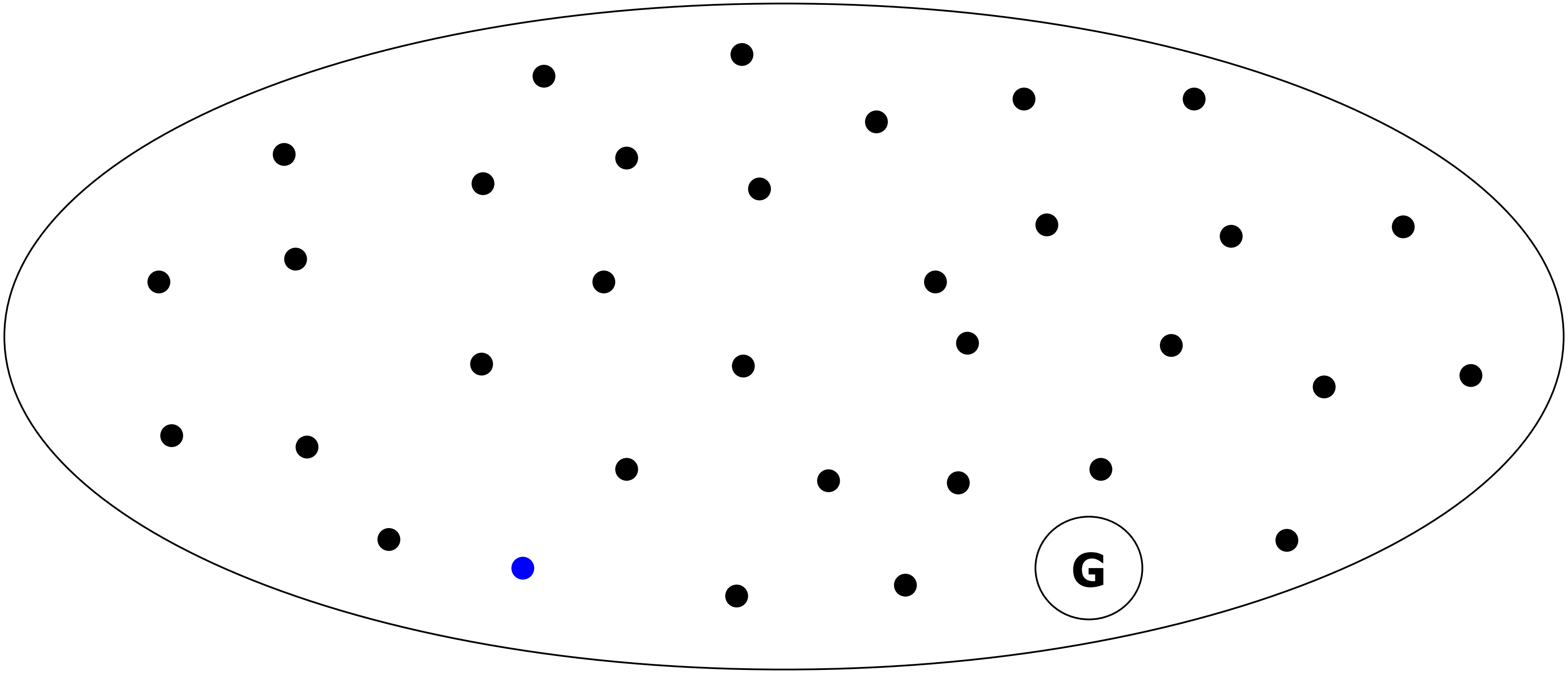
Summary: Components of a GA

A problem definition as input, and:

- Encoding principles (gene, chromosome)
- Initialization procedure (creation)
- Selection of parents (reproduction)
- Genetic operators (mutation, recombination)
- Evaluation function (environment)
- Termination condition

Review

- What does the typical search process of the methods learned so far look like?



Monte-Carlo Methods

The core of Monte-Carlo

- Monte-Carlo methods are about randomness (again ☺)
- Before we get into Monte-Carlo, we have to discuss and understand randomness a bit more …

The core of Monte-Carlo

- Randomness
- Definition of random from Merriam-Webster:
  - Main Entry: **random**
    Function: *adjective*
    Date: 1565
    1 a : lacking a definite plan, purpose, or pattern **b :** made, done, or chosen at random <read *random* passages from the book>
    2 a : relating to, having, or being elements or events with definite probability of occurrence <*random* processes> **b :** being or relating to a set or to an element of a set each of whose elements has equal probability of occurrence <a *random* sample>; *also* **:** characterized by procedures designed to obtain such sets or elements <*random* sampling>

What is the obvious problem?

Random Number

- What is random number?
- Is 3 a random number?
  – There is no such thing as single random number
- Random numbers
  – A sequence of numbers that have nothing to do with the other numbers in the sequence
    - 1,2,3,4,5,6,7,8,9,10,11,… ?
- In a uniform distribution of random numbers in the range [0,1], every number has the same chance of turning up.
  – 0.00001 is just as likely as 0.5000

Random v. Pseudo-random

- **Random numbers** have no defined sequence or formulation. Thus, for any *n* random numbers, each appears with equal probability.
- Computer algorithms are restricted to generating what we call **pseudo-random numbers**.
  - Generating a sequence of numbers whose properties approximate the properties of sequences of random numbers.
  - Based on one initial number only (the “seed”)

Creating random numbers

- Any idea for how to create random numbers with a computer?
  - (“import random” is **not** the answer ☺)

An early example (John von Neumann, 1946)

- To generate 10 digits of integer
  - Start with one of 10 digits integers
  - Square it and take middle 10 digits from answer
  - Example: $5772156649^2 = 33317\underline{7923805949}09291$
- The sequence appears to be random, but each number is determined from the previous → not random.
- Smaller example:
  - $6100^2 = 37\underline{2100}00$
  - $2100^2 = 04\underline{4100}00$
  - $4100^2 = 16\underline{8100}00$
  - …

Any problem?

An early example (John Von Neumann,1946)

- To generate 10 digits of integer
  - Start with one of 10 digits integers
  - Square it and take middle 10 digits from answer
  - Example: $5772156649^2 = 3331779238059490929$[1]
- The sequence appears to be random, but each number is determined from the previous → not random.
- Serious problem : Small numbers (0 or 1) are lumped together, it can get itself to a short loop. For example:
  - $6100^2 = 37\underline{2100}00$
  - $2100^2 = 04\underline{4100}00$
  - $4100^2 = 16\underline{8100}00$
  - $8100^2 = 65\underline{6100}00$
  - ☹

Initial number=**seed**

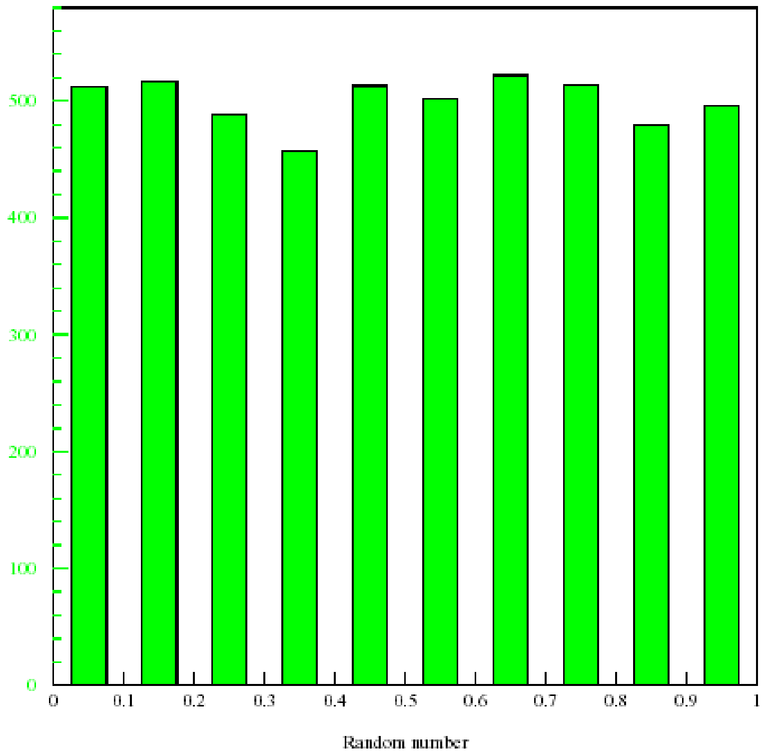
RANDU Generator

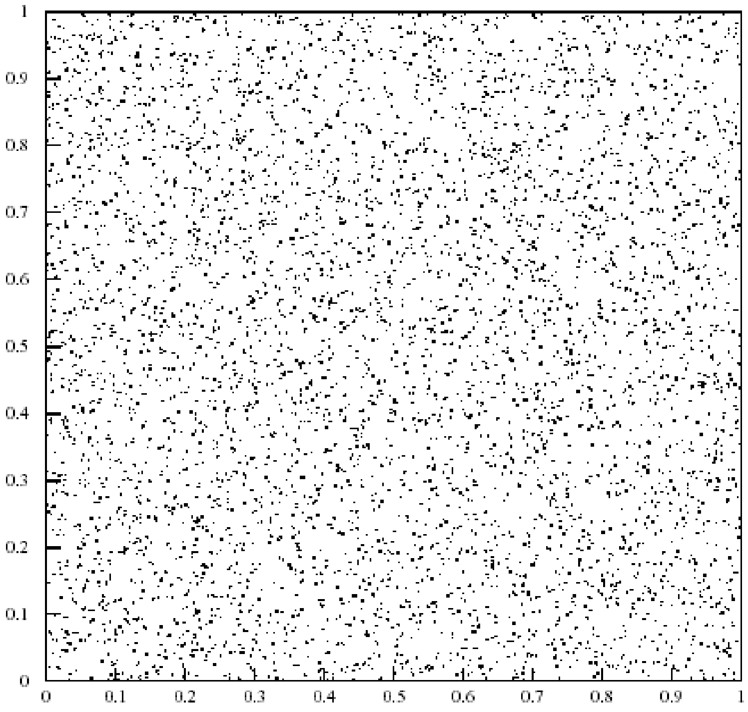
- 1960's IBM
- Algorithm:

$$I_{n+1} = (65539 \times I_n) \bmod (2^{31})$$

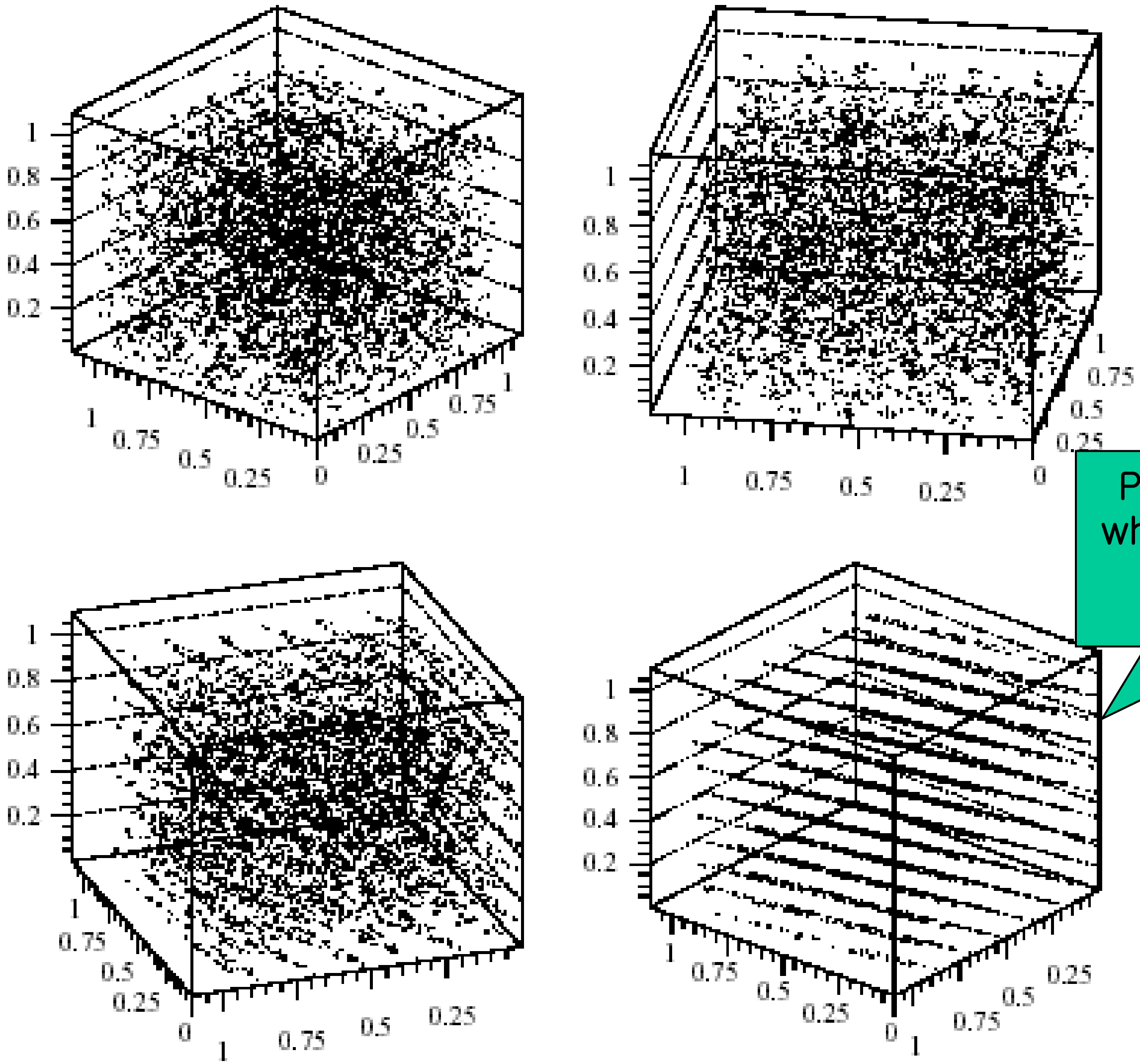
Any problem?

1D and 2D Distribution of RANDU





3D Distribution of RANDU



Problems seen only when observed at the right angle

There is a lot of research on that

- For now, just remember that pseudo-random numbers:
  1. Can be controlled by a seed
  2. Are in fact periodic sequences (e.g., when last number=seed)
  3. A finite set of quantized numbers -> problems in high-dimensions

Initializing with Seeds

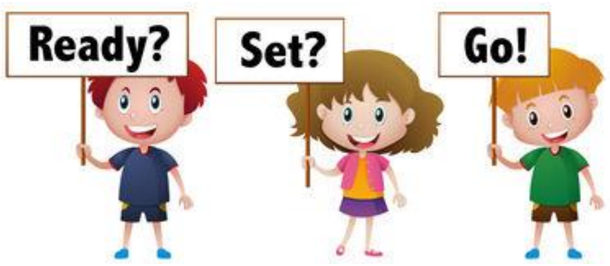
Two major reasons to initialize the seed:

1. The default state always generates the same sequence of random numbers. Not really random at all, particularly for a small set of calls. Solution: Call the seed method with the lower-order bits of the system clock.
2. You need a deterministic process that is repeatable.

Clear so far?

A small intermediate challenge

- **Problem**: What is the probability that 10 dice throws add up exactly to 32?



A small intermediate challenge

- **Problem**: What is the probability that 10 dice throws add up exactly to 32?

1. **Exact Way.** Calculate this exactly by counting all possible ways of making 32 from 10 dice.

2. **Approximate (Lazy) Way.** Simulate throwing the dice (say 500 times), count the number of times the results add up to 32, and divide this by 500.

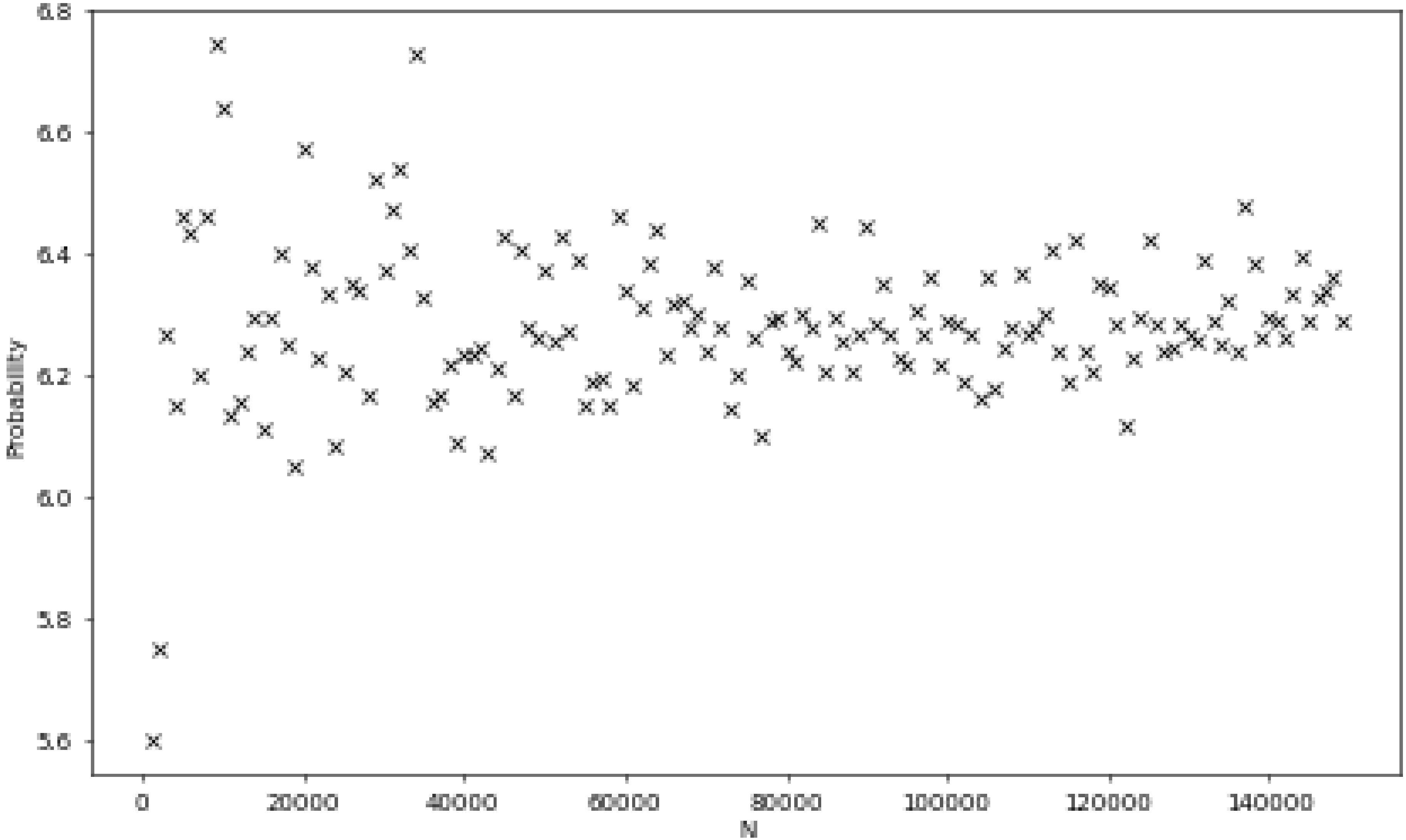
Let us do this in Python

- **Problem**: What is the probability that 10 dice throws add up exactly to 32?
- **Solution**: Let's try the approximate way

Source code for approximate simulation

```
import random
import matplotlib.pyplot as plt

fig,ax=plt.subplots(1,1,figsize=(15,10),dpi=70)
for N in range(1000,150000,1000):
    print(N)
    sum32=0
    for run in range(N):
        if sum([random.randint(1,6) for i in range(10)])==32:
            sum32+=1
    p=sum32*100/N
    ax.plot(N,p,"kx")

ax.set_xlabel("N")
ax.set_ylabel("Probability")
```

Chart for approximate simulation

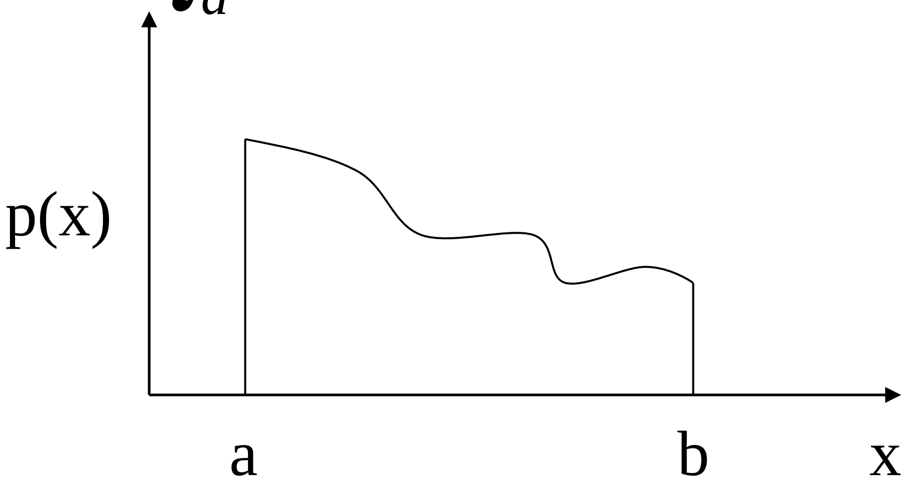


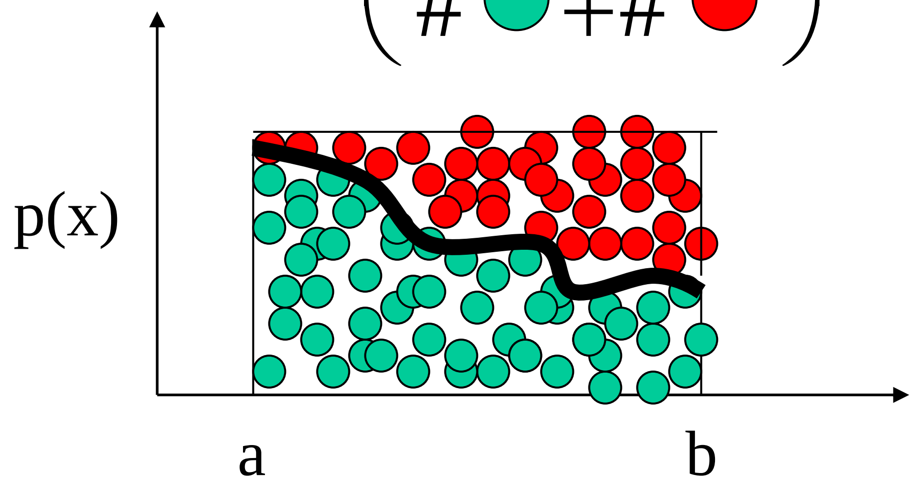
Simple Example:

$$\int_a^b p(x)dx$$

- Method 1: Analytical Integration
- Method 2: Quadrature
- Method 3: MC -- random sampling the area enclosed by a<x<b and 0<y<max (p(x))

$$\int_a^b p(x)dx \approx \max(p(x))(b-a)\left(\frac{\#\,{\color{green}\bullet}}{\#\,{\color{green}\bullet}+\#\,{\color{red}\bullet}}\right)$$





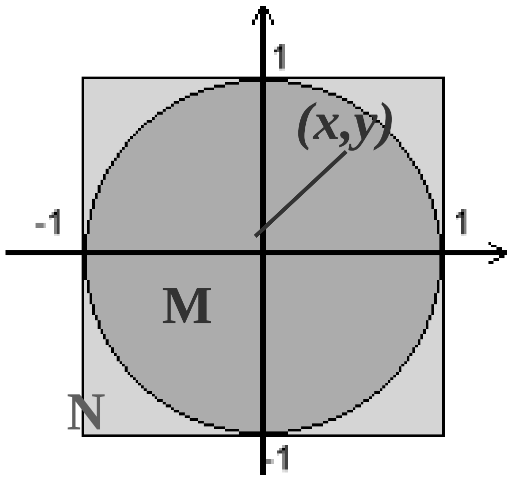
Challenge: Estimating π using Monte Carlo

- Can we estimate π using Monte-Carlo?

Estimating π using Monte Carlo

- The probability of a random point lying inside the unit circle:

$$\mathbf{P}\left(x^2 + y^2 < 1\right) = \frac{A_{circle}}{A_{square}} = \frac{\pi}{4}$$



- If pick a random point N times and M of those times the point lies inside the unit circle:

$$\mathbf{P}^{\circ}\left(x^2 + y^2 < 1\right) = \frac{M}{N}$$

$$\pi = \frac{4 \cdot M}{N}$$

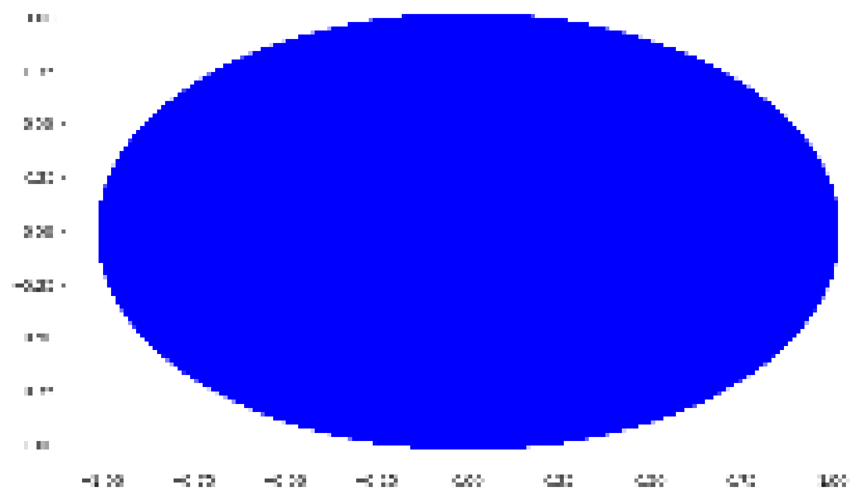
- If N becomes very large, PI=P0
- Let’s try that in Python!

Estimating π using Monte Carlo

```
import random
import math
import matplotlib.pyplot as plt

inside,outside=[],[]
for i in range(10000000):
    x=random.uniform(-1,1)
    y=random.uniform(-1,1)
    if math.sqrt(x*x+y*y)<=1:
        inside.append((x,y))
    else:
        outside.append((x,y))

print(len(inside)*4/(len(outside)+len(inside)))
fig,ax=plt.subplots(1,1,figsize=(10,6),dpi=50)

xs,ys=list(zip(*inside))
ax.plot(xs,ys,"bo")
ax.plot([x for x,y in inside],[y for x,y in inside],"bo")
```

Estimating π using Monte Carlo

- Results:
  - **N = 10,000 Pi= 3.104385**
  - **N = 100,000 Pi= 3.139545**
  - **N = 1,000,000 Pi= 3.139668**
  - **N = 10,000,000 Pi= 3.141774**
  - **...**



Question:

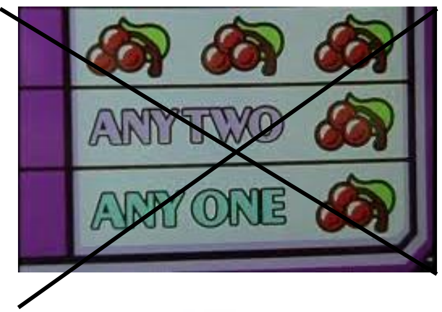
- Do we need randomness here?

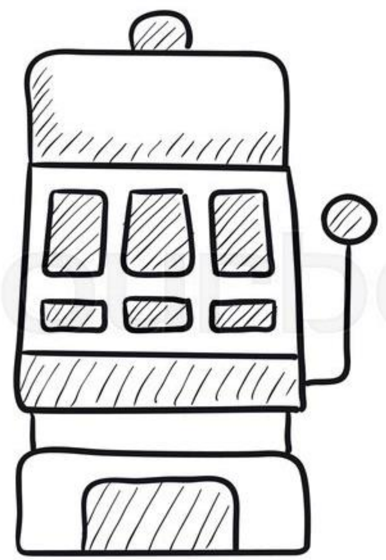
Question:

- Do we need randomness here?
- In fact, this is really a sampling problem
  - We could also evenly distribute points along the plane and compute the ratio

Multi-Armed Bandit Problem

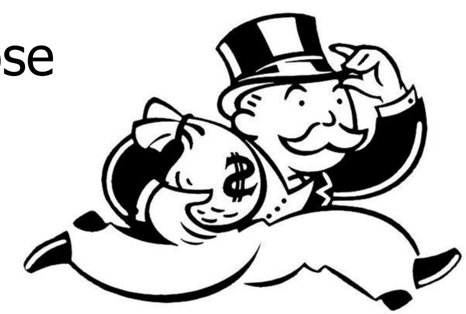
Single bandit



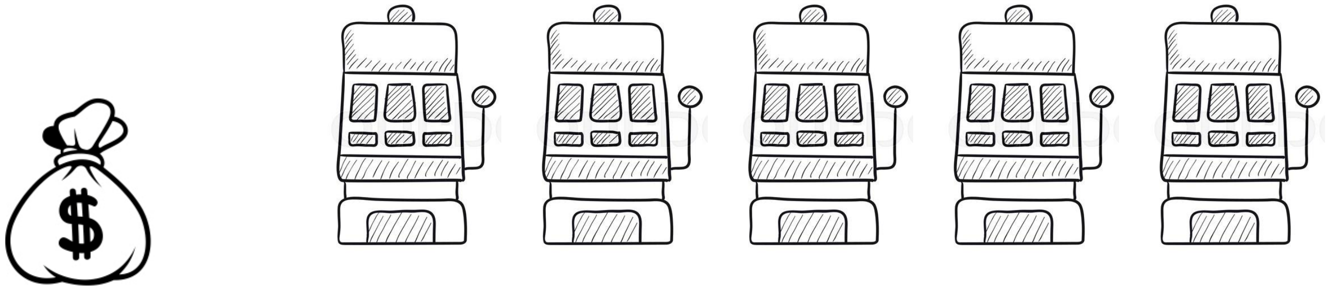


- Process:

1. Pull arm of bandit
2. Wait for three symbols to appear
   - If they are identical => win
   - If they are not identical => loose



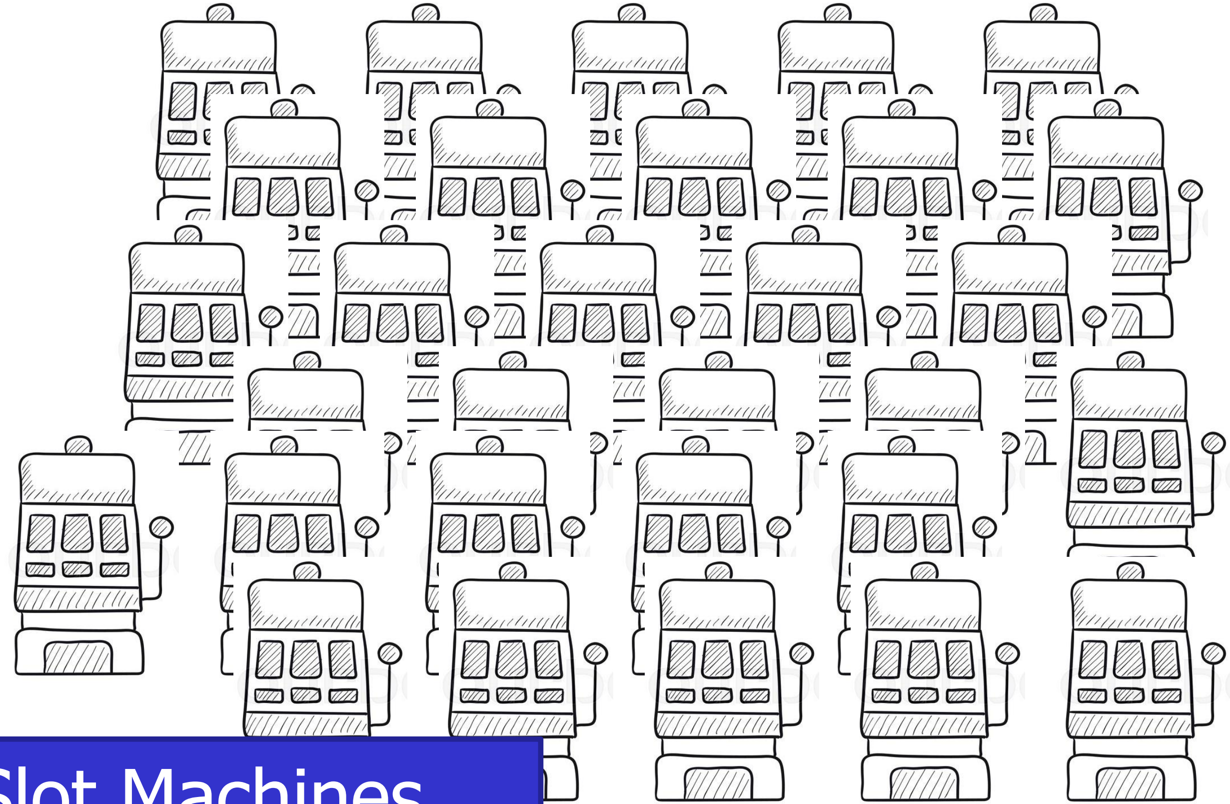
Multi-Armed Bandit Problem



No two Slot Machines are the same!

So: How to pick between Slot Machines so that you walk out with most $$$ from Las Vegas?

Problem

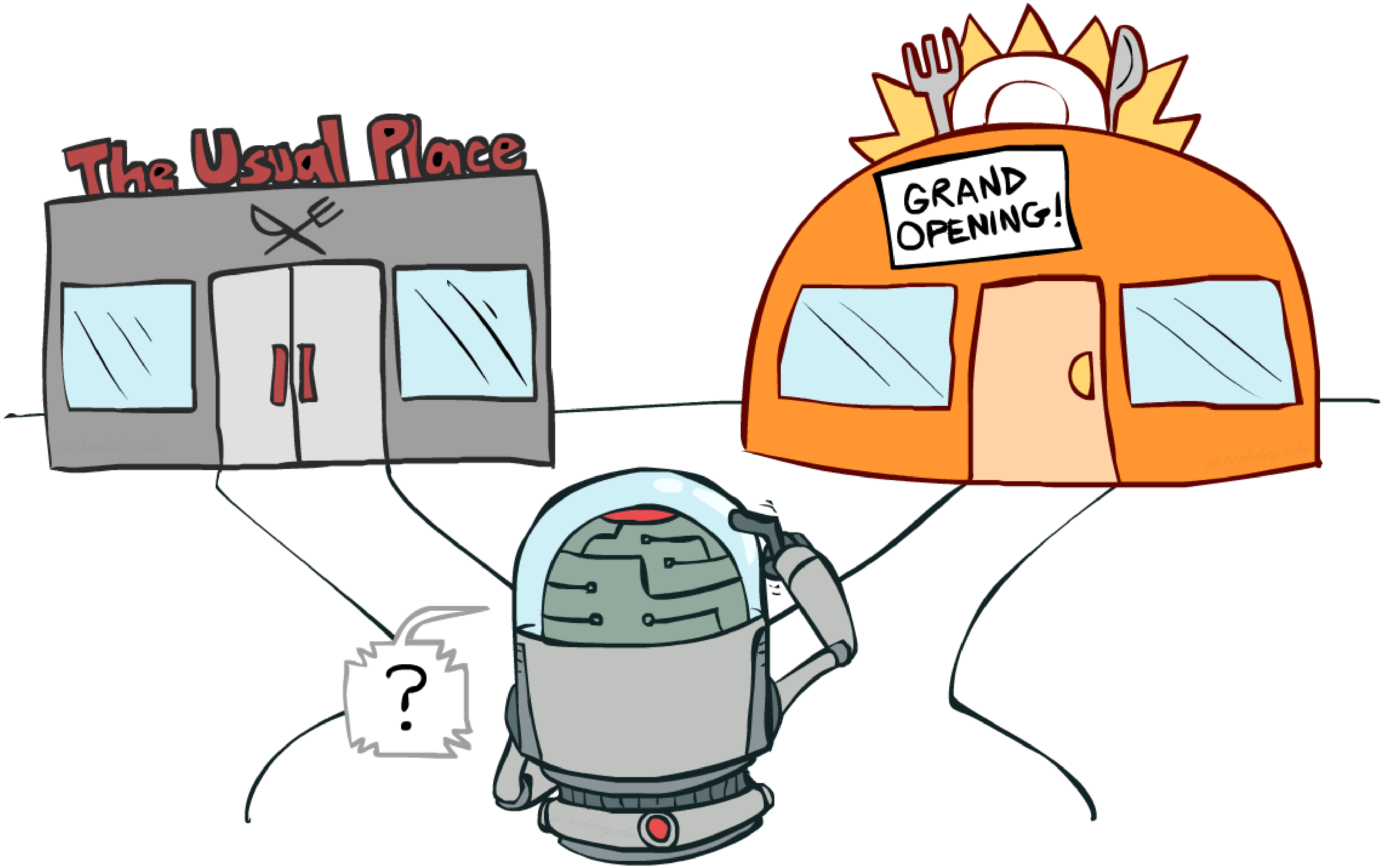


We are broke

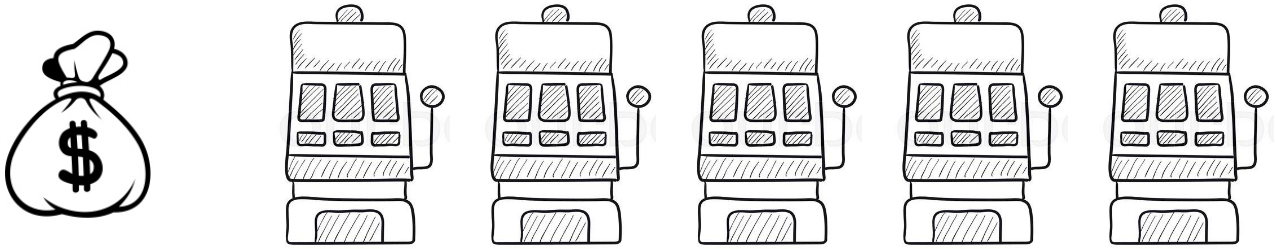
Thousands of Slot Machines

We don't have enough budget to explore each Slot Machine even once!!

An analogy in the “real world”



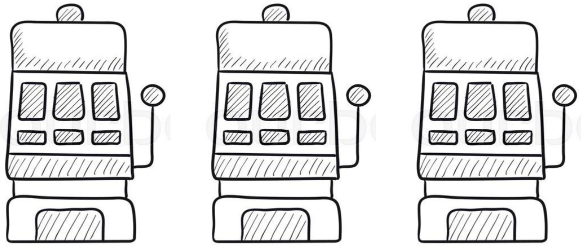
Multi-Armed Bandit Problem

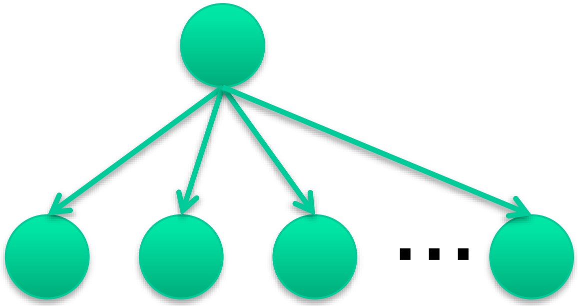


As a game?

Money = Computational Budget

Slot Machine = Next Action to Choose





Number of Slot Machines = Branching Factor

UCB Algorithm for Minimizing Cumulative Regret

- n(a) : number of pulls of arm a so far
- Q(a) : average reward for trying action a so far
- Action choice by UCB after n pulls:

$$a_n = \arg\max_a Q(a) + \sqrt{\frac{2 \ln n}{n(a)}}$$

- Assumes rewards in [0,1]. We can always normalize given a bounded reward assumption

UCB= Upper Confidence Bound

UCB: Bounded Sub-Optimality

$$a_n = \arg\max_a Q(a) + \sqrt{\frac{2\ln n}{n(a)}}$$

Value Term:
favors actions that looked good historically

Exploration Term:
actions get an exploration bonus that grows with ln(n)

Doesn't waste much time on sub-optimal arms, unlike uniform!

UCB Performance Guarantee

- Theorem: The expected cumulative regret of UCB $\boldsymbol{E}[\boldsymbol{Reg_n}]$ after n arm pulls is bounded by O(log n)

- Is this good?
- Yes. The average per-step regret is $O\left(\frac{\log(n)}{n}\right)$

- Theorem: No algorithm can achieve a better expected regret (up to constant factors)

Task

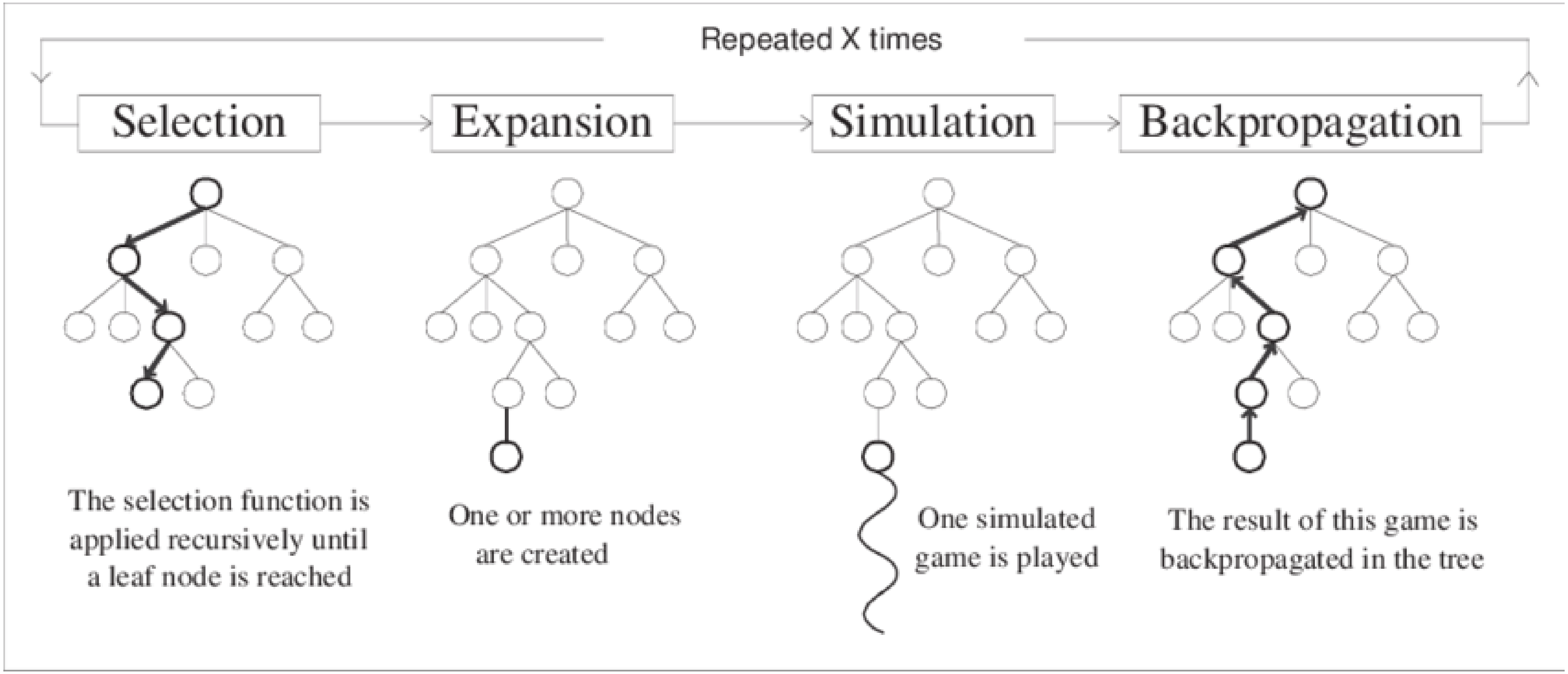
- Let us implement the multi-armed bandit problem in Python!

Generalization:

Monte-Carlo Tree Search

Monte-Carlo Tree Search

- Builds a sparse look-ahead tree rooted at current state by repeated Monte-Carlo simulation of a "**rollout policy**"



Summary

- Monte-Carlo algorithms and MCTS have revolutionized the area of game playing by computers
  - Alpha GO uses MCTS at its core
  - But the success of MCTS goes back to the early 2000s
- These algorithms have very strong theoretical properties (assuming infinite amount of time)
  - The more time you have (=the more you sample), the better results you get
- Many variants exist
  - Tuned for specific problem types

Thank you very much!