# Probabilistic Algorithms: What, Why, and How

A Deep Dive into Randomness in Computing

Sailesh Dahal

May 12, 2025

Kathmandu University

# Outline

# What are Probabilistic Algorithms?

# What is a Probabilistic Algorithm?

**Definition**
An algorithm that makes random choices during execution to influence its behavior or output.

- Output or performance may vary on different runs
- Two main types: Las Vegas (always correct, time varies), Monte Carlo (time fixed, may err)

# Types of Probabilistic Algorithms

- **Las Vegas:** Always correct, random running time
- **Monte Carlo:** Fixed running time, may give incorrect result with small probability
- **Example:** Randomized Quicksort (Las Vegas)
- **Example:** Primality testing (Monte Carlo)

# Why Probabilistic Algorithms?

# Why Randomness?

**Motivation**

- Simpler algorithms
- Better expected performance
- Avoid worst-case scenarios
- Useful for large-scale and distributed systems

# Real-World Motivation

- Web search (PageRank)
- Load balancing (power of two choices)
- Hashing (universal hash functions)
- Primality testing (Miller-Rabin)

# How do Probabilistic Algorithms Work?

# How: Randomization in Algorithms

**Key Idea**
Use random choices to influence the algorithm's path or output.

- Random pivot in Quicksort
- Random walks in graphs
- Random sampling

# Example: Randomized Quicksort

# QuickSort vs Randomized QuickSort

**QuickSort:**

1. Pick a pivot element from the array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself
3. Recursively sort the subarrays, and concatenate them

**Randomized QuickSort:**

1. Pick a pivot element **uniformly at random** from the array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself
3. Recursively sort the subarrays, and concatenate them

# Example: Randomized Quicksort

**Recall:** QuickSort can take $\Omega(n^2)$ time to sort an array of size $n$.

# Randomized QuickSort: Expected Runtime

**Theorem**

Randomized QuickSort sorts a given array of length $n$ in $O(n \log n)$ expected time.

**Note:** On every input, randomized QuickSort takes $O(n \log n)$ time in expectation. On every input, it may take $\Omega(n^2)$ time with some small probability.

# Randomized Quicksort: Step 1 (Initial Array)

Consider the array:

| 15 | 3 | 1 | 10 | 9 | 0 | 6 | 4 |
|---|---|---|---|---|---|---|---|

# Randomized Quicksort: Step 1 (Initial Array, Pivot Chosen)

Suppose the random pivot chosen is 10 (at index 3):

| 15 | 3 | 1 | 10 | 9 | 0 | 6 | 4 |
|---|---|---|---|---|---|---|---|

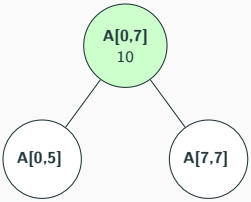
# Randomized Quicksort: Step 2 (Partitioning)

After selecting pivot 10, we partition the array:

- Left: 4, 3, 1, 9, 0, 6 (elements before pivot position)
- Middle: 10 (pivot)
- Right: 15 (element after pivot position)

After partitioning:

| 4 | 3 | 1 | 9 | 0 | 6 | 10 | 15 |
|---|---|---|---|---|---|---|---|

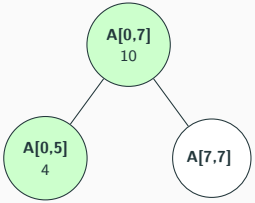


# Randomized Quicksort: Step 3 (Recurse to Left Subarray)

Recurse on the left subarray:

Let's choose a random pivot, say 4.

| 4 | 3 | 1 | 9 | 0 | 6 |
|---|---|---|---|---|---|



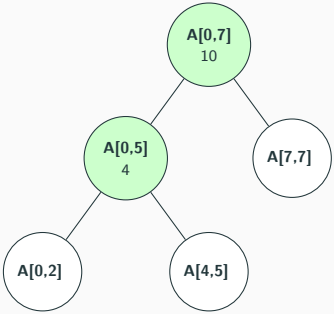
# Randomized Quicksort: Step 3 (Left Subarray Partitioning)

After partitioning the left subarray:

| 0 | 3 | 1 | 4 | 9 | 6 |
|---|---|---|---|---|---|

Partition:

- Left: 0, 3, 1 (elements before pivot)
- Middle: 4 (pivot)
- Right: 9, 6 (elements after pivot)

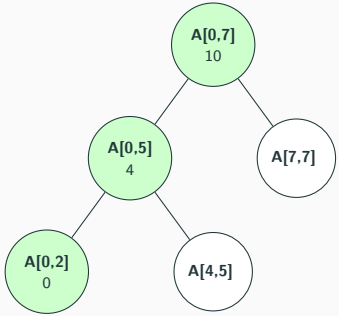


# Randomized Quicksort: Step 3 (Recurse to Left Subarray)

Recurse on the left subarray:

Let's choose a random pivot, say 0.

| 0 | 3 | 1 |
|---|---|---|



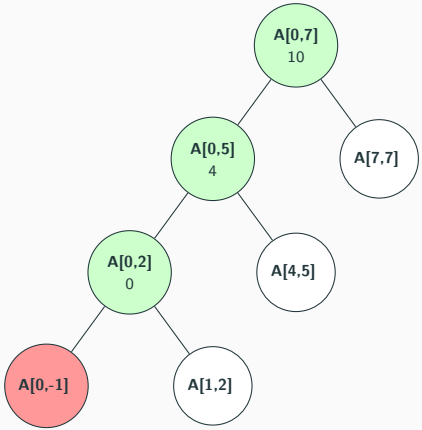
# Randomized Quicksort: Step 3 (Left Subarray Partitioning)

After partitioning the left subarray:

| 0 | 3 | 1 |
|---|---|---|

Partition:

- Left: (empty)
- Middle: 0 (pivot)
- Right: 3, 1 (elements after pivot)

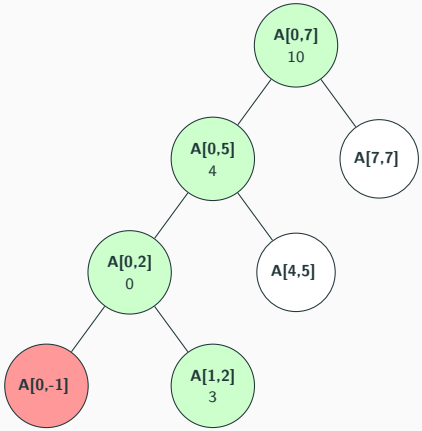


# Randomized Quicksort: Step 3a1 (Right of 0 Subarray)

Recurse on the right subarray:

Let's choose a random pivot, say 3.

| 3 | 1 |
|---|---|



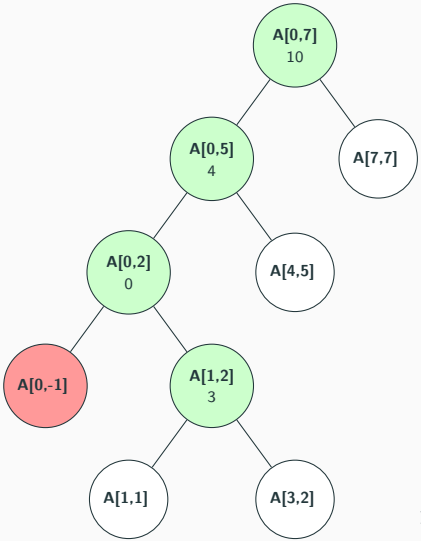
# Randomized Quicksort: Step 3 (Left Subarray Partitioning)

After partitioning the right subarray:

| 1 | 3 |
|---|---|

Partition:

- Left: 1 (element before pivot)
- Middle: 3 (pivot)
- Right: (empty)



# Randomized Quicksort: Step 3 (Left Subarray)

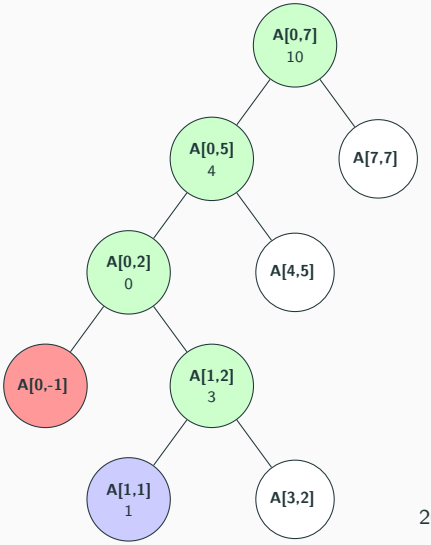
After partitioning the left subarray:

1

Partition:

- Left: (empty)
- Middle: 1 (pivot)
- Right: (empty)

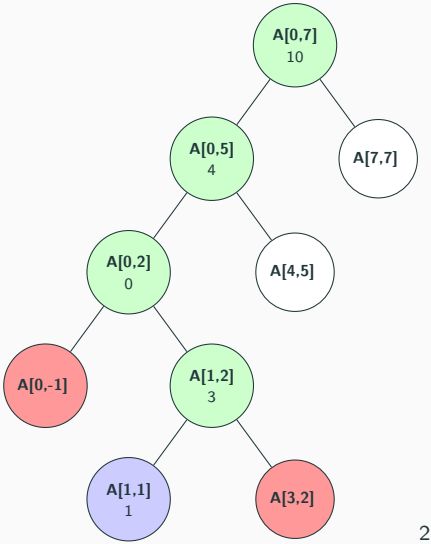
Single element subarray, done, return.



# Randomized Quicksort: Step 3a2 (Right of 3 Subarray)

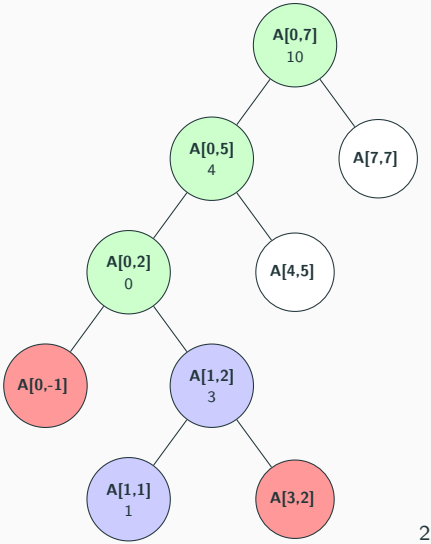
Recurse on the right subarray $A[3, 2]$ (empty, done).

Return to parent call $A[1, 2]$



# Randomized Quicksort: Step 3a2 (Right of 3 Subarray)

Return to parent call $A[0, 2]$

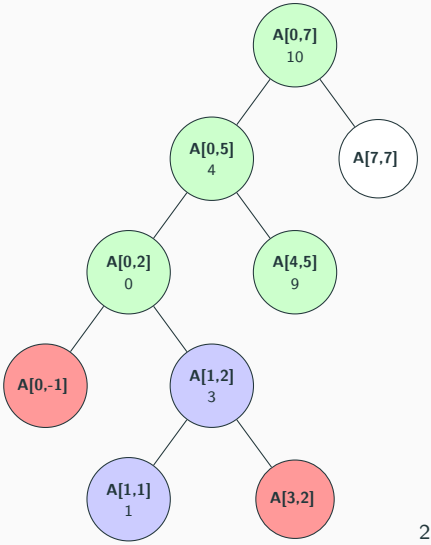


# Randomized Quicksort: Step 3a2 (Right of 3 Subarray)

Recurse on the right subarray $A[4, 5]$:

Let's choose a random pivot, say 9.

| 9 | 6 |
|---|---|



# Randomized Quicksort: Step 3b (Right of 4 Subarray)

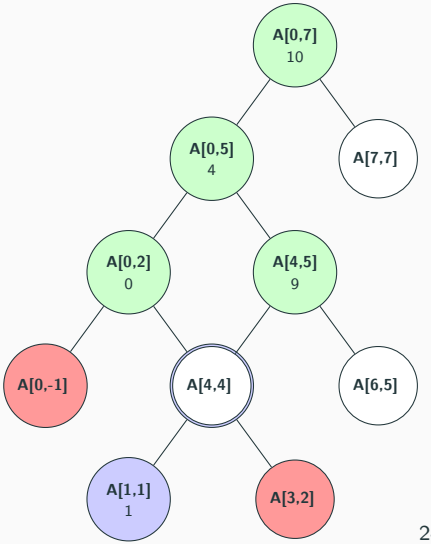
Recurse on the right subarray $A[4, 5]$:

Suppose the random pivot is 9:

| 6 | 9 |
|---|---|

Partition:

- Left: 6 (element before pivot)
- Middle: 9 (pivot)
- Right: (empty)



# Randomized Quicksort: Step 3b1 (Leftmost Subarray)

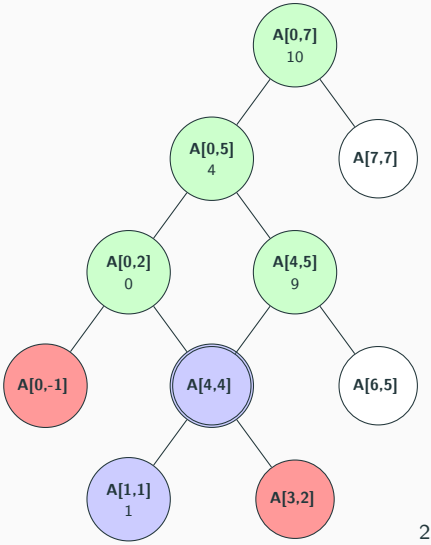
After partitioning the left subarray:

6

Partition:

- Left: (empty)
- Middle: 6 (pivot)
- Right: (empty)

Single element subarray, done, return.

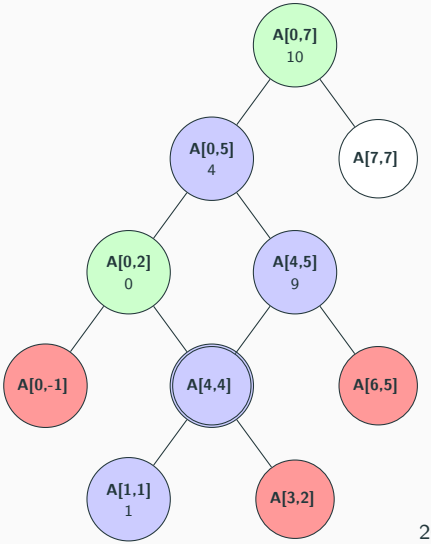


# Randomized Quicksort: Step 3b1 (Leftmost Subarray)

Recurse on the right subarray $A[6, 5]$ (empty, done).

Return to parent call $A[4, 5]$

Return to parent call $A[0, 5]$



# Randomized Quicksort: Step 3b1 (Leftmost Subarray)

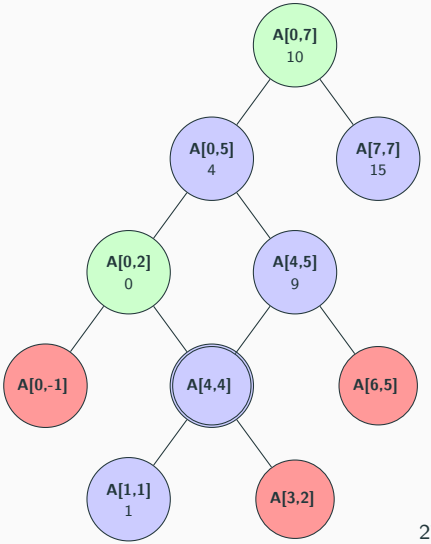
Recurse on the right subarray $A[7, 7]$ After partitioning the right subarray:

1

Partition:

- Left: (empty)
- Middle: 1 (pivot)
- Right: (empty)

Single element subarray, done, return.



# Randomized Quicksort: Final Sorted Array

The final sorted array is:

| 0 | 1 | 3 | 4 | 6 | 9 | 10 | 15 |
|---|---|---|---|---|---|---|---|

# Quicksort Implementation (Python)

```python
import random
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot_idx = random.randrange(len(arr))  # Random index
    pivot = arr[pivot_idx]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

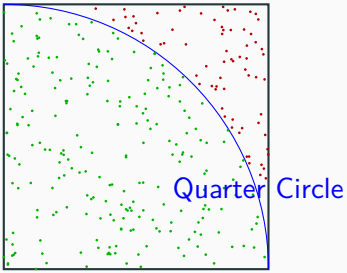
# Monte Carlo Method - Estimating $\pi$

The Monte Carlo method estimates $\pi$ by simulating random points in a unit square and counting how many fall inside a quarter circle of radius 1. The ratio of points inside the circle to the total points, multiplied by 4, approximates $\pi$.

# Monte Carlo Algorithm

1. Generate $N$ random points $(x, y)$ where $0 \leq x \leq 1$ and $0 \leq y \leq 1$.
2. For each point, check if it lies inside the quarter circle: $x^2 + y^2 \leq 1$.
3. Count the number of points $M$ that satisfy the condition.
4. Estimate $\pi$ as: $\pi \approx 4 \times \frac{M}{N}$.

# Visual Illustration



# Example Calculation

- Suppose we generate $N = 1000$ random points in the unit square
- After simulation, we count $M = 785$ points inside the quarter circle
- We estimate $\pi$ as:

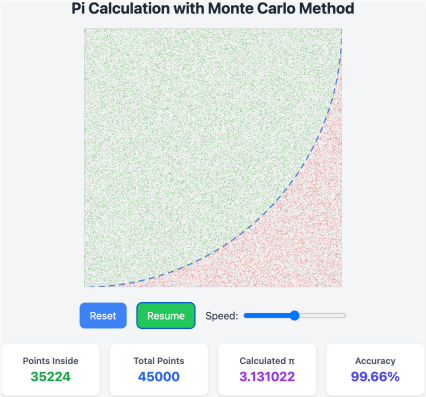
$$\pi \approx 4 \times \frac{M}{N} = 4 \times \frac{785}{1000} = 3.14$$

- The true value of $\pi$ is approximately $3.14159$

# Convergence and Error Analysis

- The error in our estimate decreases as $O(1/\sqrt{N})$
- This means:
  - $N = 100$ points gives roughly 10% error
  - $N = 10,000$ points gives roughly 1% error
  - $N = 1,000,000$ points gives roughly 0.1% error
- The Monte Carlo method is especially useful for calculating multidimensional integrals
- For $\pi$ calculation, there are more efficient methods, but this one is visually intuitive

# Demo Visualization




Open Interactive Demo

# Analysis: Recurrence and Expectation

# Quicksort Recurrence

**Expected Comparisons**

$$T(n) \leq n + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i))$$

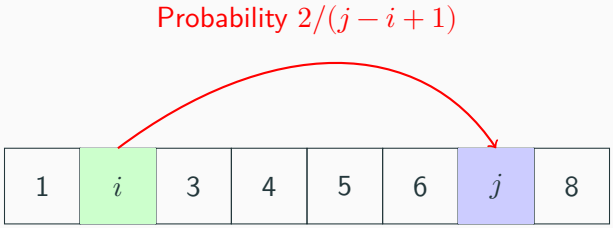
Base case: $T(1) = 0$

Solution: $T(n) = O(n \log n)$

# Slick Analysis: Indicator Variables

- $Q(A)$: Number of comparisons on input $A$
- $X_{ij}$: Indicator for whether elements $i$ and $j$ are compared
- $E[Q(A)] = \sum_{i<j} Pr[R_{ij}]$
- $Pr[R_{ij}] = \frac{2}{j-i+1}$

**Visualization**



# Harmonic Numbers in Analysis

**Harmonic Number**
$H_n = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$

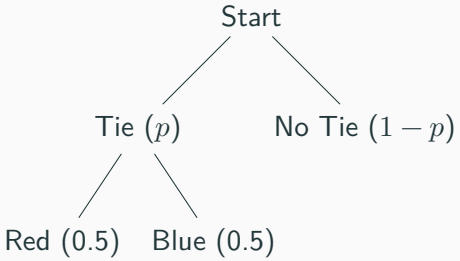
**Summation in Quicksort**
$E[Q(A)] \leq 2nH_n = O(n \log n)$

# Example: Coin Toss Probability

# Coin Toss Example

**Experiment**
John tosses a biased coin (probability $p$ of heads) to decide whether to wear a tie. If heads, tosses a fair coin to pick red or blue tie.

**Question**
What is the probability John wears a red tie on the first day he wears a tie?



# Random Bits in Practice

# Where do Random Bits Come From?

- Hardware random number generators
- Pseudo-random number generators (PRNGs)
- Physical phenomena (thermal noise, radioactive decay)
- In practice, PRNGs are sufficient for most applications

# Probabilistic Data Structures

# What is a Probabilistic Data Structure?

**Definition**
Data structures that use randomization or probabilistic techniques to achieve space or time efficiency, often allowing for small errors (e.g., false positives).

- Useful for large-scale data, streaming, or approximate answers
- Examples: Bloom filter, Count-Min Sketch, HyperLogLog

# Bloom Filter: What and Why?

**What is a Bloom Filter?**
A space-efficient, probabilistic data structure for set membership queries.

- Answers: "Is $x$ in the set?"
- May return false positives, but never false negatives
- Very compact compared to hash sets

# How Does a Bloom Filter Work?

1. Start with a bit array of $m$ bits, all set to 0
2. Use $k$ independent hash functions
3. To add an element, set $k$ bits (one per hash) to 1
4. To check membership, test if all $k$ bits are 1

# Bloom Filter Example

Suppose $m = 8$ bits, $k = 2$ hash functions, and we insert $\{\text{cat}, \text{dog}\}$.

**Bit Array After Insertion**

| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

- To check if "cat" is in the set, hash and check the corresponding bits
- If all are 1, answer is "possibly in set"; if any is 0, "definitely not in set"

# Bloom Filter: Trade-offs

- **False positives:** May say "in set" when not
- **No false negatives:** Never says "not in set" if it is
- **Space efficient:** Much smaller than explicit set
- **No deletions:** Standard Bloom filters do not support removing elements

# References i