

# Probabilistic Algorithms: What, Why, and How

A Deep Dive into Randomness in Computing

---

Sailesh Dahal

May 13, 2025

Kathmandu University



# Outline

What are Probabilistic Algorithms?

Applications of Probabilistic Algorithms

Why Probabilistic Algorithms?

How do Probabilistic Algorithms Work?

Example: Randomized Quicksort

Step-by-Step Execution

Time Complexity Analysis of Randomized Quicksort

Example: Monte Carlo Estimation of

Probabilistic Data Structures

# **What are Probabilistic Algorithms?**

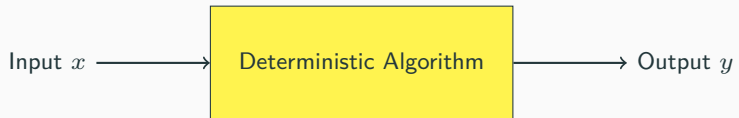
---

# What is a Probabilistic Algorithm?

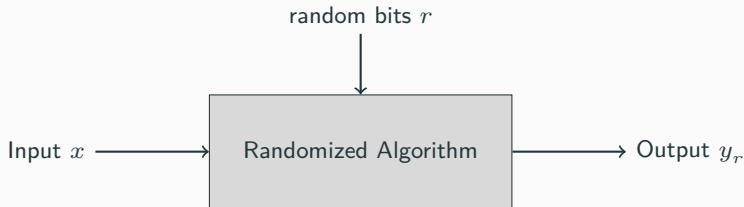
## **Definition**

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

# Deterministic vs Probabilistic Algorithm



Deterministic



Probabilistic

# Types of Probabilistic Algorithms

- Las Vegas Algorithms (Babai, 1979)
- Monte Carlo Algorithms (Metropolis & and, 1949)

**Definition:** A Las Vegas algorithm always produces a correct result or reports failure, with the running time depending on random choices. (Gupta & Ramachandran, 1992)

**Definition:** A Monte Carlo algorithm has a probability of producing an incorrect result, but its running time is bounded. (James, 1990)



# Applications of Probabilistic Algorithms

---

## Real-World Motivation

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

# Why Probabilistic Algorithms?

---

# Why Randomness?

## Motivation

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

# 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 (Hoare, 1962)
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 (Motwani & Raghavan, 1995)
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  $n$  (Sedgewick, 1978)

## Randomized QuickSort: Expected Runtime

### Theorem

Randomized QuickSort sorts a given array of length  $n$  in  $O(n \log n)$  expected time.  
(Sedgewick, 1977)

**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 1.1 (Pivot Chosen)

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

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



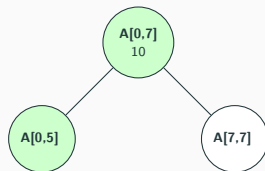
## Randomized Quicksort: Step 2 (Partitioning Around Pivot 10)

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
---	---	---	---	---	---	----	----

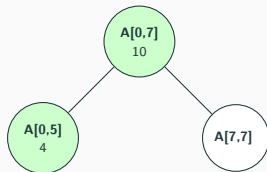


## Randomized Quicksort: Step 3 (Recurse Left [A[0,5]], Pivot 4)

Recurse on the left subarray:

Let's choose a random pivot, say 4.

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



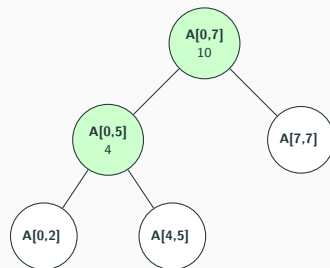
## Randomized Quicksort: Step 3.1 (Partition Left $A[0,5]$ Around 4)

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)

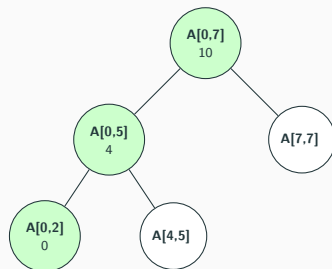


## Randomized Quicksort: Step 3.1.1 (Recurse Left [A[0,2]], Pivot 0)

Recurse on the left subarray:

Let's choose a random pivot, say 0.

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





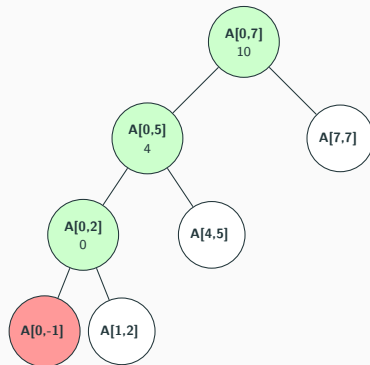
## Randomized Quicksort: Step 3.1.1.1 (Partition Left [A[0,2]] Around 0)

After partitioning the left subarray:

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

Partition:

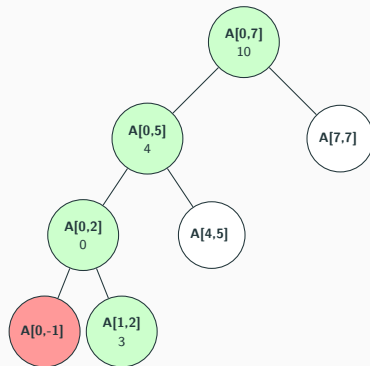
- **Left:** (empty)
- **Middle:** 0 (pivot)
- **Right:** 3, 1 (elements after pivot)



## Randomized Quicksort: Step 3.1.1.2 (Recurse Right [A[1,2]], Pivot 3)

Recurse on the right subarray:

Let's choose a random pivot, say 3.



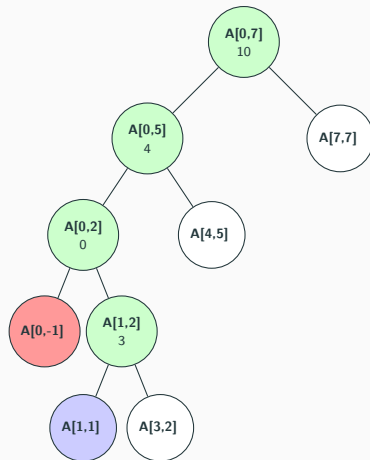
## Randomized Quicksort: Step 3.1.1.2.1 (Partition [A[1,2]] Around 3)

After partitioning the left subarray:



Partition:

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



## Randomized Quicksort: Step 3.1.1.2.1.1 (Recurse Left [A[1,1]], Done)

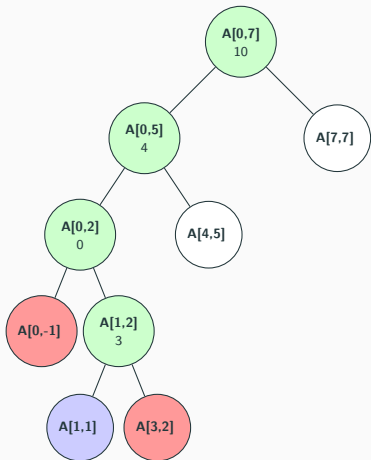
After partitioning the left subarray:

1

Partition:

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

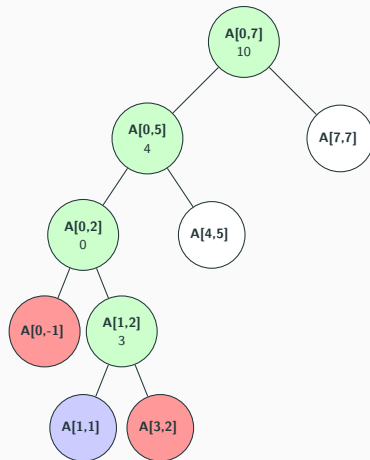
Single element subarray, done, return.



## Randomized Quicksort: Step 3.1.1.2.1.2 (Recurse Right [A[3,2]], Done)

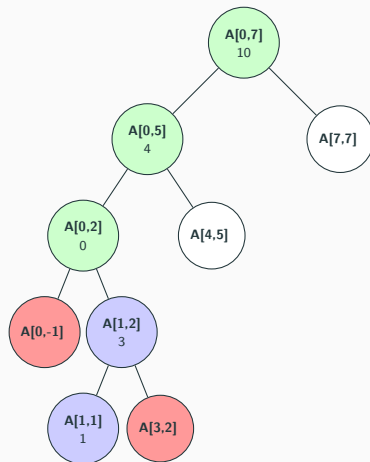
Recurse on the right subarray  $A[3, 2]$  (empty, done).

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



## Randomized Quicksort: Step 3.1.2 (Return to $A[0,2]$ )

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

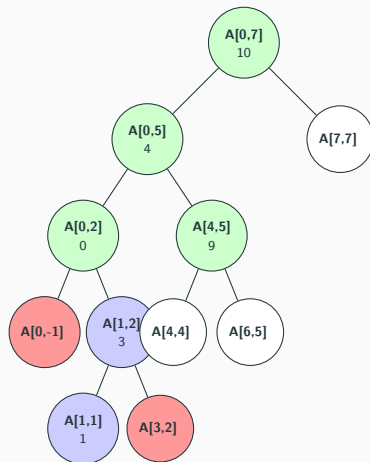


## Randomized Quicksort: Step 3.2 (Recurse Right [A[4,5]], Pivot 9)

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

Let's choose a random pivot, say 9.

9	6
---	---



## Randomized Quicksort: Step 3.2.1 (Partition $A[4,5]$ Around 9)

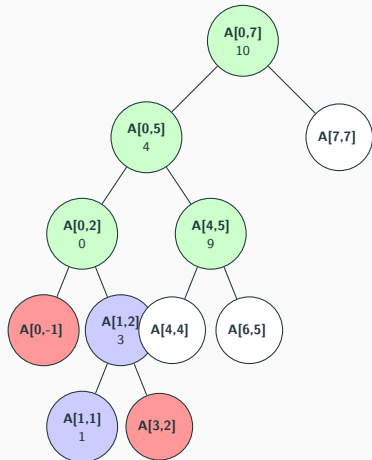
Recurse on the right subarray  $A[4, 5]$ :

Suppose the random pivot is 9:

6	9
---	---

Partition:

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





## Randomized Quicksort: Step 3.2.1.1 (Recurse Left [A[4,4]], Done)

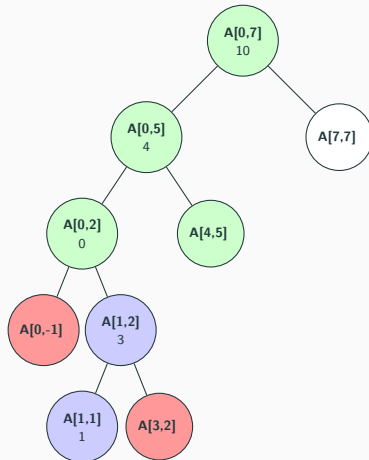
After partitioning the left subarray:

6

Partition:

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

Single element subarray, done, return.

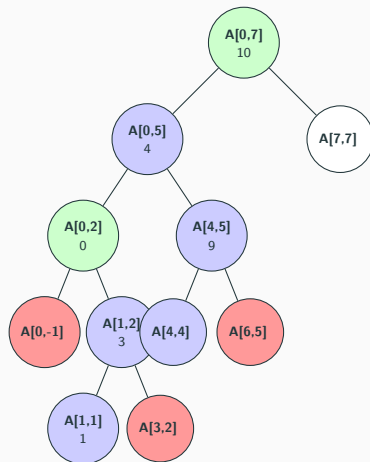


## Randomized Quicksort: Step 3.2.1.2 (Recurse Right $A[6,5]$ , Done)

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 3.3 (Recurse Right [A[7,7]], Done)

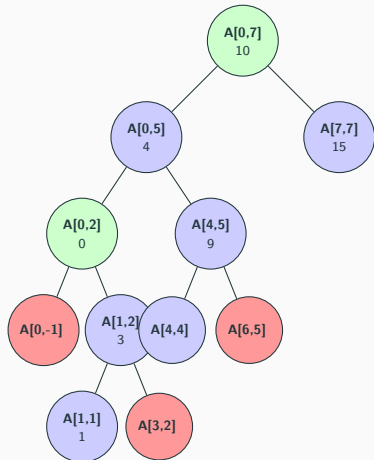
Recurse on the right subarray  $A[7,7]$  After partitioning the right subarray:

1

Partition:

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

Single element subarray, done, return.



## Randomized Quicksort: Step 4 (Final Sorted Array)

The final sorted array is:

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

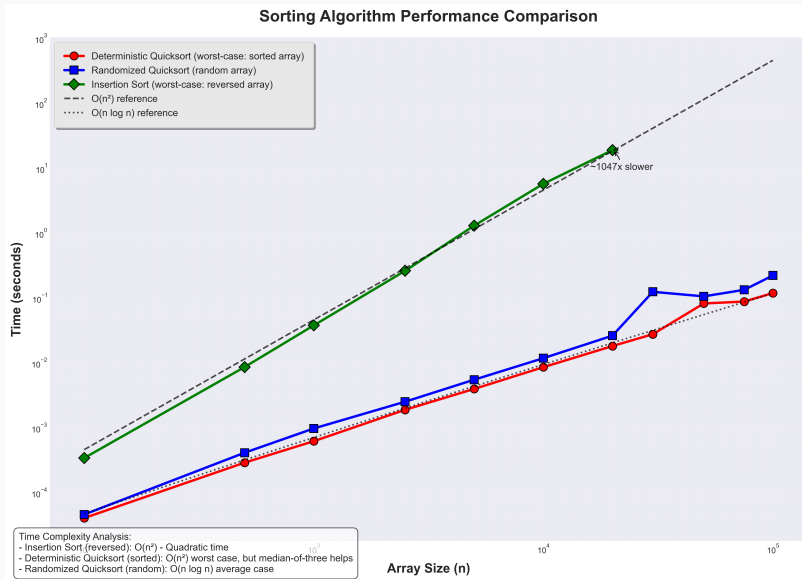
# Quicksort Time Complexity

- **Worst-case:**  $O(n^2)$
- **Best-case:**  $O(n \log n)$
- **Expected:**  $O(n \log n)$  (randomized)

# Time Complexity Analysis of Randomized Quicksort

---

# Sorting Algorithms Benchmark



# Quicksort Recurrence

## Expected Comparisons

Let  $T(n)$  be the expected number of comparisons to sort  $n$  distinct elements using randomized quicksort:

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

- $n$  comparisons in partitioning: each element compared to the pivot.
- Pivot is chosen uniformly at random.
- For pivot at position  $i$ , recursive calls sort subarrays of size  $i-1$  and  $n-i$ .
- We average over all  $n$  possible pivot positions.

## Base Case

$$T(1) = 0 \quad (\text{single element requires no comparisons})$$



# Solving the Recurrence Step-by-Step

## Step 1: Write the Recurrence

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

**Step 2: Guess the Solution** Assume  $T(n) \leq cn \log n$  for some constant  $c$ .

# Solving the Recurrence Step-by-Step

## Step 3: Plug the Guess

$$T(n) \leq n + \frac{2c}{n} \sum_{k=1}^{n-1} k \log k$$

Use:

$$\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$$

Then:

$$T(n) \leq n + cn \log n$$

## Step 4: Conclusion

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

## Example: Monte Carlo Estimation of $\pi$

---

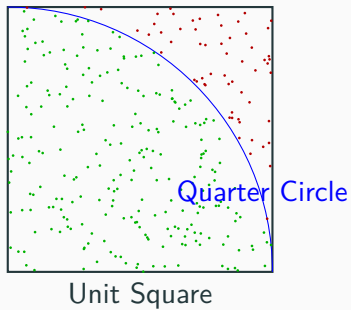
## Monte Carlo Method - Estimating $\pi$

The Monte Carlo method (Metropolis & and, 1949) 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  $\pi$  (Beckmann, 1971).

# 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:  $\pi \approx 4 \times \frac{M}{N}$  (Hammersley & Handscomb, 1964).

# Visual Illustration



1

---

<sup>1</sup>This slide was generated using a probabilistic algorithm

## 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:

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

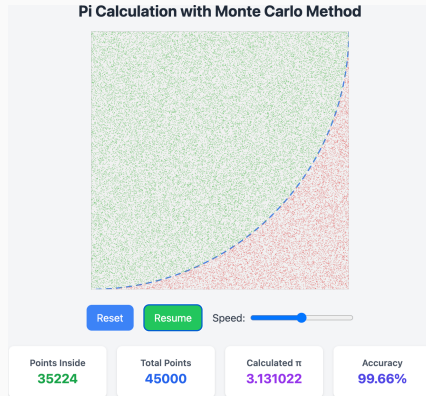
- The true value of  $\pi$  is approximately 3.14159 (Beckmann, 1971)

## Convergence and Error Analysis

- The error in our estimate decreases as  $O(1/\sqrt{N})$  (Kalos & Whitlock, 2008)
- 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 (Cookson, 2005)
- For  $\pi$  calculation, there are more efficient methods, but this one is visually intuitive



# Demo Visualization



[Open Interactive Demo](#)

# Probabilistic Data Structures

---

# Deterministic vs. Probabilistic Data Structures

## Deterministic (e.g., Hash Set, List):

- Always provide exact answers.
- Can be space-intensive (store all elements).
- Operations might be slower for large datasets (e.g., disk I/O).
- **Guarantee:** No errors (false positives or negatives).

## Key Idea

Use PDS when approximate answers are acceptable and space/speed are critical.

## Probabilistic (e.g., Bloom Filter):

- Provide approximate answers with controlled error.
- Very space-efficient (use bits, not full elements).
- Operations are typically very fast (constant time).
- **Trade-off:** Small error probability for huge efficiency gains.

## Example: Why PDS? Username Availability

### The Problem

A website with millions of users needs to instantly check if a username is available during registration. How? (Bloom, 1970)

### Deterministic Approach (Database Query):

- Store all usernames in a database.
- Query DB: 'SELECT 1 FROM users WHERE username =  $i$
- **Accurate? Yes.**
- **Fast? No.** Requires disk I/O, network latency.
- **Scalable? Poorly.** High load on DB servers.

### Probabilistic Approach (Bloom Filter):

- Keep a compact Bloom filter in memory (Bloom, 1970).
- Check filter: Is 'username' possibly present?
- **Accurate? Mostly.** Small chance of false positive (saying taken when available), needs DB check then.
- **Fast? Yes.** In-memory check is  $O(k)$ .
- **Scalable? Excellently.** Drastically reduces DB load

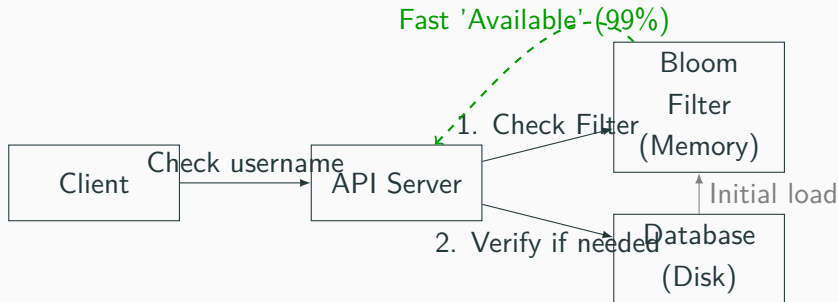
## Username Checking: Implementation Details

1. **Initialization:** Load all existing usernames into Bloom filter at service startup (only infrequent DB reads).
2. **New registrations:** Add username to both database and Bloom filter.
3. **Availability check process:**
  - Check username against Bloom filter first (Fast, in-memory)
  - If Bloom filter says "definitely not in set" → Username is available (99% case for 1% error rate)
  - If Bloom filter says "possibly in set" → Verify with database query (Slow, but rare)

### Performance Impact (10M users, 1% error)

- Memory:  $\approx$  18 MB Bloom Filter vs. hundreds of MB for DB index/cache.
- Speed: 99% of availability checks avoid slow database queries. (Wang & Reiter, 2012)

## Username Checking: System Architecture



- Bloom filter acts as a fast, efficient preliminary check.
- Deterministic check (DB) used only as a fallback.
- Massively reduces load on the expensive resource (Database).

# Bloom Filters: The Theory

- Space-efficient probabilistic data structure (Bloom, 1970)
- Tests if an element is a member of a set
- Possible false positives, never false negatives
- Components:
  - Bit array of  $m$  bits (initially all 0)
  - $k$  independent hash functions



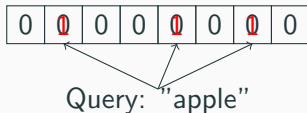
# Bloom Filter Operations

## Add element:

1. Hash element with  $k$  functions
2. Set bits at these  $k$  positions to 1

## Query element:

1. Hash element with  $k$  functions
2. Check bits at these  $k$  positions
3. If **any** bit is 0: **Definitely not in set**
4. If **all** bits are 1: **Probably in set**





# The Math Behind Bloom Filters

- **False positive probability ( $p$ ):**

$$p \approx (1 - e^{-kn/m})^k \quad (1)$$

(Broder & Mitzenmacher, 2003)

- **Optimal size ( $m$  bits) for  $n$  items, error  $p$ :**

$$m = -\frac{n \ln p}{(\ln 2)^2} \quad (2)$$

- **Optimal hash functions ( $k$ ):**

$$k = \frac{m}{n} \ln 2 \approx 0.7 \cdot \frac{m}{n} \quad (3)$$

## Time and Space Complexity

Structure	Space	Lookup	Insert	Error Type
Hash Set	$O(n)$	$O(1)$ avg	$O(1)$ avg	None
Bloom Filter	$O(m)$	$O(k)$	$O(k)$	False Positives
Sorted List	$O(n)$	$O(\log n)$	$O(n)$	None
Trie	$O(N)$	$O(L)$	$O(L)$	None

$n$ =items,  $m$ =bits ( $m \ll n \times item\_size$ ),  $k$ =hashes,  $N$ =total chars,  $L$ =key length

# Bloom Filter Simulation

## Bloom Filter Simulation

Check if your username or password has been compromised

### Check Value

Enter username or password to check:

CheckAdd to Database

Possibly compromised! This value may exist in the database.

Confidence:

93.91%

Real match probability: 93.91%

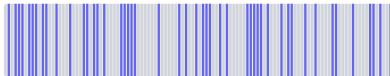
False positive probability: 6.09%

A Bloom filter can have false positives but no false negatives.

### Bloom Filter Visualization

Size: 384 bits

Hash Functions: 3



#### Settings

Filter Size



Hash Functions



Reset Filter

#### Known Compromised Values

3	admin	qwerty	123456	letmein	baseball	dragon	football	monke
2	zaq1zaq1	hello123	charlie	jesus	ninja	mustang	chocolate	starw

### About Bloom Filters

A Bloom filter is a space-efficient probabilistic data structure designed to quickly test whether an element is present in a set. It can have false positives (incorrectly reporting an element is in the set) but no false negatives (if it reports an element is not in the set, it definitely is not).

#### How It Works

1. Multiple hash functions are applied to the input element
2. Each hash function gives a position in the bit array
3. When adding an element, all corresponding bits are set to 1
4. When checking, if all corresponding bits are 1, the element might be in the set
5. If any bit is 0, the element is definitely not in the set

#### Use Cases

- Checking if a username is taken before querying a database
- "Have I Been Pwned" password checking
- Spell checkers
- Web cache sharing
- Network packet routing

# Other Applications of Bloom Filters

## Web/Database:

- Cache hit/miss optimization (e.g., CDNs)
- Avoid unnecessary DB lookups (like username example)
- Recommendation systems (seen items) (Broder & Mitzenmacher, 2003)

## Network:

- Web crawler URL deduplication (avoid re-crawling)
- Network packet routing (track flows efficiently)
- P2P network resource discovery

## Security:

- Malware signature detection
- Spam filtering (known bad IPs/domains)
- Password breach checking (HaveIBeenPwned)

## Big Data:

- Stream deduplication (unique visitors/events)
- Distributed data sync (approximate differences)
- Genomics (k-mer counting)

# When to Use Bloom Filters

Bloom filters are ideal when:

- Memory is a critical constraint (Big Data, embedded systems)
- False positives are acceptable (can be handled by a secondary check)
- False negatives are unacceptable (must find all true positives)
- Elements are expensive to store or compare
- Lookup speed is crucial (real-time systems)
- Deletions are not needed (or use variants like Counting Bloom Filters)

# References

---



Babai, L. (1979). **Monte-carlo algorithms in graph isomorphism testing.**

*Technical Report DMS, 79-10.*



Beckmann, P. (1971). ***A history of pi.*** St. Martin's Press.



Bloom, B. H. (1970). **Space/time trade-offs in hash coding with allowable errors.**

*Commun. ACM, 13(7), 422–426.* <https://doi.org/10.1145/362686.362692>



Broder, A., & Mitzenmacher, M. (2003).

**Network Applications of Bloom Filters: A Survey.**

*Internet Mathematics, 1(4), 485–509.*



Cookson, J. A. (2005). **Monte carlo error analyses of spallation neutron source**

**and water moderator.** *Nuclear Science and Engineering, 150(3), 296–314.*

<https://doi.org/10.13182/NSE05-A2527>



Gupta, S. S., & Ramachandran, R. (1992).

**A las vegas algorithm for sorting by comparison.**

*Information Processing Letters*, 41(2), 77–82.

[https://doi.org/10.1016/0020-0190\(92\)90132-4](https://doi.org/10.1016/0020-0190(92)90132-4)



Hammersley, J. M., & Handscomb, D. C. (1964). **Monte carlo methods.** Methuen.



Hoare, C. A. R. (1962). **Quicksort.** *The Computer Journal*, 5(1), 10–16.

<https://doi.org/10.1093/comjnl/5.1.10>



James, F. (1990).

**Determining the statistical accuracy of monte carlo method results.**

*Computer Physics Communications*, 60(3), 329–344.

[https://doi.org/10.1016/0010-4655\(90\)90039-9](https://doi.org/10.1016/0010-4655(90)90039-9)



Kalos, M. H., & Whitlock, P. A. (2008). **Monte carlo methods.** Wiley-VCH.



Metropolis, N., & S. U. (1949). **The monte carlo method.**

*Journal of the American Statistical Association*, 44(247), 335–341.

<https://doi.org/10.1080/01621459.1949.10483310>



Motwani, R., & Raghavan, P. (1995). **Randomized algorithms.**

Cambridge University Press.



Sedgewick, R. (1977). **The analysis of quicksort programs..** *Acta Inf.*, 7, 327–355.

<http://dblp.uni-trier.de/db/journals/acta/acta7.html#Sedgewick77>



Sedgewick, R. (1978). **Implementing quicksort programs.** *Commun. ACM*, 21(10),

847–857. <https://doi.org/10.1145/359619.359631>



Wang, X., & Reiter, M. K. (2012).

**Defending against denial-of-information attacks in social networks.**

*Proceedings of the 19th ACM Conference on Computer and Communications Security*,

63–74. <https://doi.org/10.1145/2382196.2382206>