Reservoir Sampling — Detailed Guide (k = 1 and general k)

A clear, step-by-step explanation with proofs, dry-runs, code, and notes (Hinglish-friendly)

Author: Generated for Gupta Ji (by ChatGPT)

Date: 29 August 2025

1. Introduction

Reservoir Sampling ek algorithm hai jo humein allow karta hai ki hum ek data stream (ya linked list) se uniformly at random sample lein, jab total size pehle se nahi pata ho ya jab itna bada ho ki pura memory mein store karna feasible na ho.
Is document mein hum detail se samjhenge: k = 1 case (single random element), algorithm, why it works (proof), multiple dry-runs, Java implementation with line-by-line explanation, general k case (sample of size k), complexity, trade-offs, aur kuch extra notes (including the important "== 0" note). Hinglish mein bhi easy explanations diye gaye hain taaki tum jaldi samajh jao.

2. Problem Statement

- Tumhe ek stream of elements (ya singly linked list) se exactly 1 element uniformly at random select karna hai.
- Constraint: tum stream ka total size (n) nahi jaante, aur tum zyada memory use nahi karna chahte. Sirf O(1) extra memory use karni hai.

3. Algorithm (k = 1)

Algorithm (simple form):
1) Maintain ek variable **result** (initially null or -1) and a counter **count** = 1.
2) Traverse stream/linked list element-by-element. For current element (i-th seen), generate a random integer in **[0..i-1]** i.e. rand.nextInt(i).
3) If the random integer equals a chosen fixed value (conventionally **0**), then set **result = current element**.
4) Continue until stream ends. Return **result**.

Code-convention: Java uses rand.nextInt(count) which returns integers from 0 to count-1. The check == 0 gives current element a probability of 1/count to be picked.

4. Java Implementation (k = 1) with explanation

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    ListNode head;
    Random rand;
```

```
    public Solution(ListNode head) {
        this.head = head;
        rand = new Random();
    }

    public int getRandom() {
        int result = -1;       // store chosen value
        int count = 1;         // count of nodes seen so far
        ListNode temp = head;
        while (temp != null) {
            if (rand.nextInt(count) == 0) {   // give current element 1/count chance
                result = temp.val;
            }
            temp = temp.next;
            count++;
        }
        return result;
    }
}
```

4.1 Line-by-line Explanation

- **ListNode head; Random rand;** : head points to the linked list start; rand is the random generator.
- **Solution(ListNode head)** : store head and initialize Random.
- **getRandom()** : traverse the list with a counter **count**. For each node, call rand.nextInt(count).
- For the i-th node (when count == i), rand.nextInt(i) yields 0 with probability 1/i. If that happens, replace result.
- Because of the replacement rule and multiplicative survival probabilities for earlier elements, at the end each element has probability 1/n to be the result.

5. Important Note: Why we check == 0 (or any fixed number)

- **rand.nextInt(count)** returns a uniform value from 0 to count-1 (each with probability 1/count).
- We need the current element to be selected with probability **1/count**. Therefore we simply check if the random value equals any particular fixed value (for example 0).
- So rand.nextInt(count) == 0 is just a convenient way to give current element exactly 1/count chance.
- **Important**: It does not have to be 0 specifically. You could check == 1 or == count-1 — probability remains 1/count. Conventionally people use == 0 because it's concise and common in codebases.

6. Proof Sketch: Why every element ends up with probability 1/n

We prove by induction that after processing i elements, each one of the first i elements is chosen with probability 1/i.
Base (i = 1): first element chosen with probability 1 (trivial).
Inductive step: assume after i-1 elements each has probability 1/(i-1). For the i-th element, we pick it with probability 1/i (because rand.nextInt(i) == 0 with prob 1/i).
For any previous element x (1 <= x <= i-1), probability it remains chosen after i-th step = P(x chosen after i-1 steps) * P(not replaced at i-th step) = (1/(i-1)) * (1 - 1/i) = (1/(i-1)) * ((i-1)/i) = 1/i.
Thus all i elements have probability 1/i. By induction the claim holds until n, so each final element has 1/n chance.

7. Dry Runs (step-by-step examples)

Example A: [10, 20, 30, 40] (n = 4)
- Step1 (count=1): choose 10 (prob=1).
- Step2 (count=2): 20 chosen with prob 1/2 (else 10 remains).

- Step3 (count=3): 30 chosen with prob 1/3.
- Step4 (count=4): 40 chosen with prob 1/4.
Final probabilities: each of 10,20,30,40 = 1/4.

Example B: [5,7,9] (n = 3) => each ends up with 1/3.

Example C: [100] => only element, probability 1.

Example D: [1,2,3,4,5] (n = 5) => after math each element prob = 1/5 (see multiplicative survival in proof).

Example E: Stream [A,B,C,D,E,F] (n = 6) => first element A survives with probability product of not-being-replaced = 1/2 * 2/3 * 3/4 * 4/5 * 5/6 = 1/6. Similarly every element gets 1/6.

8. General case: Selecting k samples (Reservoir of size k)

Agar humein k > 1 items chahiye uniformly at random without replacement from a stream, to algorithm thoda change hota hai:
1) Maintain an array 'reservoir' of size k. Fill it with first k elements of the stream.
2) For i = k+1, k+2, ... (i is 1-based index of current element): generate random integer r in [0..i-1].
- If r < k, replace reservoir[r] with current element.
3) Continue until stream ends. At the end, reservoir contains k items, each subset of size k equally likely.

Why it works (intuition): each new element i has probability k/i to enter the reservoir (because r < k occurs with prob k/i). Replacement logic preserves uniformity via induction and symmetry arguments.

8.1 Java Implementation (k-size reservoir)

```
import java.util.*;

class ReservoirSamplerK<T> {
    private final T[] reservoir;
    private final Random rand;
    private int count; // total items seen

    @SuppressWarnings("unchecked")
    public ReservoirSamplerK(int k) {
        reservoir = (T[]) new Object[k];
        rand = new Random();
        count = 0;
    }

    // feed next item from stream
    public void add(T item) {
        count++;
        if (count <= reservoir.length) {
            reservoir[count - 1] = item; // fill initial reservoir
        } else {
            int r = rand.nextInt(count); // [0..count-1]
            if (r < reservoir.length) {
                reservoir[r] = item;
            }
        }
    }

    public T[] getReservoir() { return reservoir; }
```

```
}
```

9. Complexity & Trade-offs

- Time: O(n) to process n elements (must inspect each element at least once).
- Space: O(1) for k=1; O(k) for general k.
- Trade-off: Reservoir Sampling saves memory compared to storing all elements, useful for huge/unknown streams. If you can afford O(n) memory and plan to call getRandom many times, you may convert to array once (O(n) space) and answer future queries in O(1) time each.

10. When to use Reservoir Sampling

- Stream size unknown or too large to store in memory.
- Need one (or k) uniform random samples from a stream without replacement.
- Memory-critical environments, e.g., big data preprocessing, online algorithms.

11. Extra Notes & Tips (Yuvraj style, Hinglish-friendly)

- "== 0" koi magic nahi — simply ek convenient fixed ticket number hai. Tum "== count-1" bhi laga sakte ho.
- Agar linked list chhoti ho aur multiple getRandom() calls chahiye, consider copying values in an array once, then pick random index each time.
- For reproducibility in tests, seed the Random object: rand = new Random(42);
- Java's rand.nextInt(count) is 0-based; dhyaan rakhna (1-based probability thinking -> compare with count variable).
- Always test with multiple trials (simulation) to see approximate uniformity with large number of runs.

12. Resources (YouTube videos)

1) https://youtu.be/9vC4I5sKFsw?si=sO7msEXig5IqJKNA
2) https://youtu.be/DWZqBN9efGg?si=iPT6dxLFPtbmH5de

These two videos explain reservoir sampling with visuals — included as requested.

13. Short Summary

- Reservoir Sampling is the standard way to pick k (or 1) items uniformly at random from a stream when n is unknown or large.
- For k = 1, use if (rand.nextInt(count) == 0) to give current element 1/count chance.
- For general k, maintain initial reservoir of size k and replace randomly with probability k/i for i-th element.
- Space-efficient and simple to implement.

End of document. Happy coding! — Generated for Gupta Ji