CS2241 Assignment 1

Nico Fidalgo

March 26, 2025

1 Problem 1: PageRank and HITS Algorithm Analysis

1.1 Problem Statement

We are given a directed graph represented by the following adjacency matrix.

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{1}$$

We need to calculate the PageRank scores and Hub and Authority scores. The interpretation of the adjacency matrix is:

- Node a has outgoing edges to nodes c, d, and f
- Node b has outgoing edges to nodes c and f
- Node c has outgoing edges to nodes d and f
- Node d has outgoing edges to nodes b and c
- Node e has an outgoing edge to node a
- Node f has an outgoing edge to node e

1.2 PageRank Algorithm

PageRank was designed to model the behavior of a random surfer on the web. The algorithm assigns a score to each node in a directed graph based on its structural importance. The intuition is that a node is important if many important nodes point to it and the importance of a node is divided among its outgoing links.

$$\mathbf{p} = (1-d) \cdot \frac{\mathbf{1}}{n} + d \cdot M \cdot \mathbf{p} \tag{2}$$

where:

- d is the damping factor (typically 0.85)
- n is the number of nodes
- $\mathbf{1}$ is a vector of all 1's
- M is the transition matrix (columns sum to 1)

For each node i with outgoing links, we set:

$$M_{j,i} = \frac{1}{\text{out-degree}(i)} \tag{3}$$

if there is a link from node i to node j, and 0 otherwise. The algorithm for computing PageRank scores is:

Algorithm 1 PageRank Score Calculation

```
Initialize $\mathbf{p}^{(0)} = \frac{1}{n} \cdot \mathbf{1}$
for $t = 0, 1, 2, \ldots$ until convergence do
    $\mathbf{p}^{(t+1)} = (1-d) \cdot \frac{\mathbf{1}}{n} + d \cdot M \cdot \mathbf{p}^{(t)}$
    if $\|\mathbf{p}^{(t+1)} - \mathbf{p}^{(t)}\| <$ tolerance then
        break
    end if
end for
Normalize $\mathbf{p}$ to sum to 1
```

1.3 PageRank Implementation

I wrote a JavaScript program to implement the PageRank algorithm:

```
import * as math from 'mathjs';

// Adjacency matrix from the problem
const A = [
    [0, 0, 1, 1, 0, 1],  // a -> *
    [0, 0, 1, 0, 0, 1],  // b -> *
    [0, 0, 0, 1, 0, 1],  // c -> *
    [0, 1, 1, 0, 0, 0],  // d -> *
    [1, 0, 0, 0, 0, 0],  // e -> *
    [0, 0, 0, 0, 1, 0]   // f -> *
];

// Number of nodes
const n = A.length;

// Out-degrees
```

```
const out_degrees = A.map(row => row.reduce((sum, val) => sum + val, 0));
console.log("Out-degrees:", out_degrees);

// Create transition matrix (column-stochastic)
const M = Array(n).fill().map(() => Array(n).fill(0));
for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
        if (A[i][j] > 0) {
            M[j][i] = 1.0 / out_degrees[i];
        }
    }
}

// PageRank parameters
const d = 0.85;  // Damping factor
const max_iter = 100;
const tol = 1e-6;

// Initialize PageRank
let pr = Array(n).fill(1/n);

// Algorithm iteration
for (let iter = 0; iter < max_iter; iter++) {
    // Calculate M * pr
    const M_pr = Array(n).fill(0);
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            M_pr[i] += M[i][j] * pr[j];
        }
    }

    // Calculate (1-d)/n + d * (M * pr)
    const pr_new = M_pr.map(val => (1-d)/n + d * val);

    // Check convergence
    const diff = math.norm(pr_new.map((val, idx) => val - pr[idx]));
    if (diff < tol) {
        pr = pr_new;
        console.log(‘\nPageRank converged after ${iter+1} iterations.‘);
        break;
    }

    pr = pr_new;
}

// Normalize to sum to 1
const pr_sum = pr.reduce((sum, val) => sum + val, 0);
pr = pr.map(val => val / pr_sum);
```

1.4 PageRank Results

After running the algorithm, I obtained the following PageRank scores:

Node	PageRank Score
a	0.186551
b	0.091079
c	0.182643
d	0.155479
e	0.190060
f	0.194188

Nodes f, e, and a have the highest PageRank scores, indicating they are structurally important in this network. Node b has the lowest score, suggesting it's less central in the graph's link structure.

1.5 HITS Algorithm

The Hyperlink-Induced Topic Search (HITS) algorithm identifies two types of important nodes in a directed graph: hubs which are nodes that point to many good authorities, and authorities which are nodes that are pointed to by many good hubs. The hub ($\mathbf{h}$) and authority ($\mathbf{a}$) vectors satisfy:

$$\mathbf{a} = A^T\mathbf{h} \tag{4}$$

$$\mathbf{h} = A\mathbf{a} \tag{5}$$

where A is the adjacency matrix of the graph. The algorithm for computing HITS scores is:

Algorithm 2 HITS Score Calculation

```
Initialize $\mathbf{h}^{(0)} = \mathbf{1}$ and $\mathbf{a}^{(0)} = \mathbf{1}$
for $t = 0, 1, 2, \ldots$ until convergence do
    $\mathbf{a}^{(t+1)} = A^T\mathbf{h}^{(t)}$
    Normalize $\mathbf{a}^{(t+1)}$
    $\mathbf{h}^{(t+1)} = A\mathbf{a}^{(t+1)}$
    Normalize $\mathbf{h}^{(t+1)}$
    if $\|\mathbf{a}^{(t+1)} - \mathbf{a}^{(t)}\| <$ tolerance and $\|\mathbf{h}^{(t+1)} - \mathbf{h}^{(t)}\| <$ tolerance then
        break
    end if
end for
```

1.6 HITS Implementation

Again, I wrote a JavaScript program to implement the HITS algorithm:

```
import * as math from 'mathjs';

// Adjacency matrix from the problem
const A = [
    [0, 0, 1, 1, 0, 1],  // a -> *
    [0, 0, 1, 0, 0, 1],  // b -> *
    [0, 0, 0, 1, 0, 1],  // c -> *
    [0, 1, 1, 0, 0, 0],  // d -> *
    [1, 0, 0, 0, 0, 0],  // e -> *
    [0, 0, 0, 0, 1, 0]   // f -> *
];

// Number of nodes
const n = A.length;

// HITS parameters
const max_iter = 100;
const tol = 1e-6;

// Initialize hub and authority scores
let hub = Array(n).fill(1);
let auth = Array(n).fill(1);

// Compute transpose of A
const AT = Array(n).fill().map(() => Array(n).fill(0));
for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
        AT[i][j] = A[j][i];
    }
}

// HITS iteration
for (let iter = 0; iter < max_iter; iter++) {
    // Update authority scores: a = A^T * h
    const auth_new = Array(n).fill(0);
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            auth_new[i] += AT[i][j] * hub[j];
        }
    }

    // Normalize authority scores
    const auth_norm = math.norm(auth_new);
    const auth_normalized = auth_new.map(val => val / auth_norm);

    // Update hub scores: h = A * a
    const hub_new = Array(n).fill(0);
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            hub_new[i] += A[i][j] * auth_normalized[j];
        }
    }

    // Normalize hub scores
    const hub_norm = math.norm(hub_new);
    const hub_normalized = hub_new.map(val => val / hub_norm);

    // Check convergence
    const auth_diff = math.norm(auth_normalized.map((val, idx) => val - auth[idx]));
    const hub_diff = math.norm(hub_normalized.map((val, idx) => val - hub[idx]));

    if (auth_diff < tol && hub_diff < tol) {
        auth = auth_normalized;
        hub = hub_normalized;
        console.log(‘HITS converged after ${iter+1} iterations.‘);
        break;
    }

    auth = auth_normalized;
    hub = hub_normalized;
}
```

1.7 HITS Results

After running the HITS algorithm, I obtained the following scores:

Node	**Hub Score**	**Authority Score**
a	0.684439	0.000000
b	0.501536	0.113935
c	0.446890	0.590796
d	0.283360	0.454889
e	0.000000	0.000000
f	0.000000	0.656548

Nodes a, b, and c have high hub scores, indicating they are good at pointing to authority nodes. This makes sense as a and b point to multiple nodes including high authority nodes c and f. Nodes e and f have zero hub scores because they don't point to any nodes with high authority scores.

Nodes f and c have the highest authority scores, followed by node d. This means they are pointed to by good hub nodes. Again, this makes sense as node f and c are pointed to by nodes a and b which have high hub scores. Nodes a and e have zero authority scores because they aren't pointed to by any good hub nodes.

1.8 Assumptions

For PageRank scores:

- Used a damping factor of 0.85 (standard value)
- Defined convergence as when L2 norm difference $< 10^{-6}$

For HITS:

- Defined convergence as when L2 norm difference $< 10^{-6}$

2 Problem 2: Random Walk Methods for Hub and Authority Scores

2.1 Problem Statement

We need to prove that the Hub score for a page is proportional to the number of outlinks and that the Authority score is proportional to the number of inlinks when using the following random walk approach:

For Authority scores:

- From page p_1, follow back a random inlink to page p_2
- From p_2, follow forward a random outlink to page p_3
- The step takes us from p_1 to p_3

For Hub scores:

- From page p_1, follow forward a random outlink to page p_2
- From p_2, follow backward a random inlink to page p_3
- The step takes us from p_1 to p_3

We assume the Markov chains are finite, irreducible, and aperiodic, ensuring a unique stationary distribution.

2.2 Definitions

- $A[i,j] = 1$ if there's a link from page i to page j, 0 otherwise
- $\text{in}(j) =$ number of inlinks to page $j = \sum_i A[i,j]$
- $\text{out}(i) =$ number of outlinks from page $i = \sum_j A[i,j]$

2.3 Transition Probabilities

For Authority Random Walk:
When starting at page i:

- The probability of following a random inlink back to page k is $\frac{A[k,i]}{\text{in}(i)}$
- The probability of following a random outlink from k to j is $\frac{A[k,j]}{\text{out}(k)}$

Therefore, the transition probability from i to j is:

$$P_a(i,j) = \sum_k \frac{A[k,i]}{\text{in}(i)} \times \frac{A[k,j]}{\text{out}(k)} \tag{6}$$

For Hub Random Walk:
When starting at page i:

- The probability of following a random outlink to page k is $\frac{A[i,k]}{\text{out}(i)}$
- The probability of following a random inlink back from k to j is $\frac{A[j,k]}{\text{in}(k)}$

Therefore, the transition probability from i to j is:

$$P_h(i,j) = \sum_k \frac{A[i,k]}{\text{out}(i)} \times \frac{A[j,k]}{\text{in}(k)} \tag{7}$$

2.4 Authority Score Proof

Let's hypothesize that the stationary distribution $\pi_a(i)$ is proportional to in(i), i.e., $\pi_a(i) = c \times \text{in}(i)$ for some constant c.

For this to be a stationary distribution, it must satisfy:

$$\pi_a(j) = \sum_i \pi_a(i) P_a(i,j) \tag{8}$$

Substituting our hypothesis:

$$\begin{align} \pi_a(j) &= \sum_i c \times \text{in}(i) \times \sum_k \frac{A[k,i]}{\text{in}(i)} \times \frac{A[k,j]}{\text{out}(k)} \tag{9} \\ &= c \times \sum_i \sum_k \frac{A[k,i] \times A[k,j]}{\text{out}(k)} \tag{10} \\ &= c \times \sum_k \frac{A[k,j]}{\text{out}(k)} \times \sum_i A[k,i] \tag{11} \end{align}$$

Since $\sum_i A[k,i] = \text{out}(k)$ (the number of outlinks from page k):

$$\begin{align} \pi_a(j) &= c \times \sum_k \frac{A[k,j]}{\text{out}(k)} \times \text{out}(k) \tag{12} \\ &= c \times \sum_k A[k,j] \tag{13} \\ &= c \times \text{in}(j) \tag{14} \end{align}$$

This confirms the hypothesis that the Authority score $\pi_a(j)$ is proportional to in(j), the number of inlinks to page j.

2.5 Hub Score Proof

Similarly, hypothesize that the stationary distribution $\pi_h(i)$ is proportional to out(i), i.e., $\pi_h(i) = d \times \text{out}(i)$ for some constant d.

For this to be a stationary distribution, it must satisfy:

$$\pi_h(j) = \sum_i \pi_h(i) P_h(i,j) \tag{15}$$

Substituting our hypothesis:

$$\begin{aligned}
\pi_h(j) &= \sum_i d \times \text{out}(i) \times \sum_k \frac{A[i,k]}{\text{out}(i)} \times \frac{A[j,k]}{\text{in}(k)} && (16) \\
&= d \times \sum_i \sum_k \frac{A[i,k] \times A[j,k]}{\text{in}(k)} && (17) \\
&= d \times \sum_k \frac{A[j,k]}{\text{in}(k)} \times \sum_i A[i,k] && (18)
\end{aligned}$$

Since $\sum_i A[i,k] = \text{in}(k)$ (the number of inlinks to page k):

$$\begin{aligned}
\pi_h(j) &= d \times \sum_k \frac{A[j,k]}{\text{in}(k)} \times \text{in}(k) && (19) \\
&= d \times \sum_k A[j,k] && (20) \\
&= d \times \text{out}(j) && (21)
\end{aligned}$$

This confirms the hypothesis that the Hub score $\pi_h(j)$ is proportional to $\text{out}(j)$, the number of outlinks from page j.

3 Problem 3: Reservoir Sampling

3.1 Single-Item Reservoir Sampling

We need to prove that the following algorithm maintains a uniform sample of one item among all items seen so far:

- When the first item appears, store it in memory
- When the k-th item appears, replace the current item with probability $1/k$

I will prove by induction that after processing k items, each item has exactly $1/k$ probability of being in memory. In the base case where $k = 1$, after seeing the 1st item, it is stored with probability 1. Since we've only seen one item, this gives us a uniform distribution. Beyond the base case, assume that after seeing $k-1$ items, each item has a probability of $1/(k-1)$ of being in memory. Now the k-th item arrives. For the k-th item, the probability it replaces the current item is $1/k$, therefore, $P(k\text{-th item in memory}) = 1/k$. For any previous item i (where $1 \le i < k$):

$$
\begin{align}
P(i\text{-th item in memory after } k \text{ items}) &= P(i\text{-th item was in memory}) \times P(\text{it remains in memory}) \tag{22}\\
&= \frac{1}{k-1} \times \left(1 - \frac{1}{k}\right) \tag{23}\\
&= \frac{1}{k-1} \times \frac{k-1}{k} \tag{24}\\
&= \frac{1}{k} \tag{25}
\end{align}
$$

Thus, after processing the k-th item, each of the k items seen so far has exactly $1/k$ probability of being in memory, which confirms the uniform distribution property.

3.2 Sampling s Items Without Replacement

Now we generalize to maintaining a uniform sample of s items without replacement. The algorithm is:

1. Store the first s items as they arrive
2. When the k-th item arrives (where $k > s$):
   - With probability s/k, include it in the sample
   - If including it, replace a randomly chosen item from the current sample

I need to prove that after seeing k items ($k \geq s$), each item has exactly s/k probability of being in the sample. In the base case, after seeing s items, all s items are in the sample with probability 1. Since $s/s = 1$, this is uniform. Beyond the base case, assume that after seeing $k-1$ items ($k-1 \geq s$), each has probability $s/(k-1)$ of being in the sample. Now the k-th item arrives. We need to show that after processing it, each of the k items has probability exactly s/k of being in the sample. For the k-th item, it's included with probability s/k, so $P(k\text{-th item in sample}) = s/k$. For any previous item i (where $1 \leq i < k$):

$$\begin{aligned} P(i\text{-th item remains in sample}) &= P(i\text{-th item was in sample}) \\ &\quad \times P(i\text{'s not replaced}) && (26) \\ &= \frac{s}{k-1} \times [1 - P(k\text{-th item is chosen}) \\ &\quad \times P(\text{item } i \text{ is replaced} \mid k\text{-th is chosen})] && (27) \\ &= \frac{s}{k-1} \times \left[1 - \frac{s}{k} \times \frac{1}{s}\right] && (28) \\ &= \frac{s}{k-1} \times \left[1 - \frac{1}{k}\right] && (29) \\ &= \frac{s}{k-1} \times \frac{k-1}{k} && (30) \\ &= \frac{s}{k} && (31) \end{aligned}$$

This confirms that after processing the k-th item, each of the k items has probability s/k of being in the sample, maintaining uniformity.

3.3 Implementation

To verify these theoretical results, I implemented both algorithms in JavaScript and ran simulations to check if the sampling is uniform.

```javascript
// Single-item reservoir sampling
function singleItemReservoirSampling(stream) {
    let sample = null;
    let count = 0;

    for (const item of stream) {
        count++;
        if (count === 1) {
            sample = item;
        } else {
            // Replace with probability 1/count
            if (Math.random() < 1/count) {
                sample = item;
            }
        }
    }

    return sample;
}

// Reservoir sampling of s items without replacement
function reservoirSampling(stream, s) {
    let sample = [];
    let count = 0;

    for (const item of stream) {
        count++;

        if (count <= s) {
            // Fill the reservoir with the first s items
            sample.push(item);
        } else {
            // Decide whether to include this item in the sample
            const j = Math.floor(Math.random() * count);
            if (j < s) {
                // Replace the randomly chosen item
                sample[j] = item;
            }
        }
    }

    return sample;
}

// Function to simulate multiple runs and track results
function runSimulations(numRuns, streamSize, sampleSize = 1) {
    const counts = {};

    // Initialize counts for each number
    for (let i = 1; i <= streamSize; i++) {
        counts[i] = 0;
    }

    for (let run = 0; run < numRuns; run++) {
        // Create a stream of numbers from 1 to streamSize
        const stream = Array.from({length: streamSize}, (_, i) => i + 1);

        let result;
        if (sampleSize === 1) {
            result = [singleItemReservoirSampling(stream)];
        } else {
            result = reservoirSampling(stream, sampleSize);
        }

        // Update counts
        result.forEach(item => {
            counts[item]++;
        });
    }

    // Convert to frequencies
    const frequencies = {};
    for (const [key, value] of Object.entries(counts)) {
        frequencies[key] = value / numRuns;
    }

    return frequencies;
}
```

3.4 Results

I ran 10,000 simulations with a stream of 10 items for both single-item sampling and sampling 3 items without replacement. The results confirm our theoretical analysis:

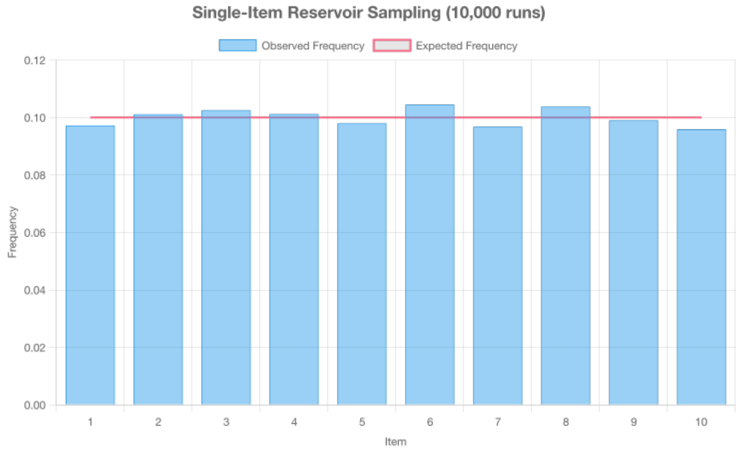

Figure 1: Frequency distribution of single-item reservoir sampling over 10,000 runs. Each item has approximately 0.1 probability of being selected, as expected.

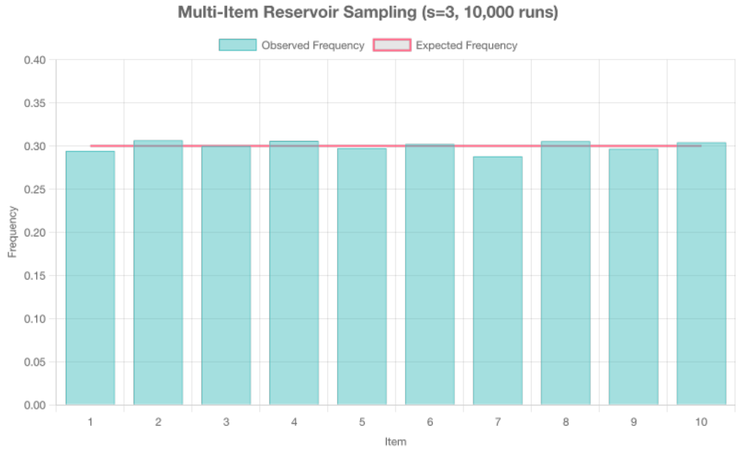

Figure 2: Frequency distribution of 3-item reservoir sampling over 10,000 runs. Each item has approximately 0.3 probability of being selected, as expected.

For the single-item sampling, the expected frequency is $1/10 = 0.1$ for each item. The empirical results in Figure 1 closely match this expectation, with minimal deviation from the expected value. For the 3-item sampling, the expected frequency is $3/10 = 0.3$ for each item. Again, the empirical results in Figure 2 closely match the expectation, confirming our theoretical analysis. These simulations confirm the theoretical correctness of both algorithms, showing that they indeed maintain uniform sampling of the stream at each step.

3.5 Why This Is Useful in Streaming Algorithms

Reservoir sampling is particularly valuable in streaming algorithms because it maintains a representative sample without needing knowledge of the total stream size, it uses constant memory as it stores only s items, it processes each item in constant time, and the sample remains uniform at each step of the process. This makes it ideal for applications like processing large data streams that don't fit in memory, online analytics where a representative sample is needed in real time, distributed systems where data is generated across multiple sources, and monitoring high-volume log data where sampling is necessary for performance.

4 Problem 4: Flipping a Coin Until First Head

4.1 Problem Statement

A fair coin is flipped until the first head occurs. Let X be the number of flips required. I must first find the entropy $H(X)$ in bits and then describe how to ask yes/no questions to determine X adn find the expected number of such questions and compare this result to $H(X)$.

4.2 (a) Entropy Calculation

Since the coin is fair, the probability that $X = k$ (i.e., the first head occurs on the k-th flip) is

$$P(X = k) \;=\; \left(\frac{1}{2}\right)^k, \quad k = 1, 2, 3, \ldots$$

We use the fact that for a geometric random variable with parameter $p = \frac{1}{2}$ (counting the number of trials up to and including the first success), the distribution is:

$$P(X = k) = \frac{1}{2}\left(\frac{1}{2}\right)^{k-1} = \left(\frac{1}{2}\right)^k.$$

The entropy in bits is

$$H(X) \;=\; -\sum_{k=1}^{\infty} P(X = k)\, \log_2\bigl[P(X = k)\bigr].$$

Substitute $P(X = k) = (1/2)^k$, and note that $\log_2\left[(1/2)^k\right] = -k$. So

$$H(X) = -\sum_{k=1}^{\infty} \left(\frac{1}{2}\right)^k (-k) = \sum_{k=1}^{\infty} k \left(\frac{1}{2}\right)^k.$$

We now use the series identity:

$$\sum_{k=1}^{\infty} k\, x^k \;=\; \frac{x}{(1-x)^2}, \quad \text{for } -1 < x < 1.$$

With $x = \frac{1}{2}$, we get

$$\sum_{k=1}^{\infty} k \left(\frac{1}{2}\right)^k = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} \;=\; \frac{\frac{1}{2}}{(\frac{1}{2})^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2.$$

Thus, the entropy $H(X) = 2$ bits.

4.3 (b) Yes/No Questions and Comparison

To determine X from yes/no questions, we want a strategy that minimizes the expected number of questions. We can think of this as constructing a decision tree for the geometric distribution.

One possible line of reasoning is:

- First question: "Is $X = 1$?"
  - With probability 1/2, the answer is yes, and we are done with just 1 question.
  - With probability 1/2, the answer is no, in which case we continue to ask about $X = 2$, $X = 3$, etc.
- We can create a decision tree that accounts for the probabilities $(1/2)^k$ at each branch.

This geometric distribution achieves an expected number of questions exactly 2, matching the entropy $H(X) = 2$ bits. In other words, on average, we need 2 yes/no questions to determine X. This matches the lower bound given by $H(X)$, illustrating that our questioning strategy is asymptotically optimal.

Hence, the entropy $H(X)$ provides a fundamental limit on the average number of bits (yes/no questions) required, and our constructed question strategy can match that limit.