Chapter 9. Medians and Order Statistics

Joon Soo Yoo

April 17, 2025

Assignment

- Read §9, §10.0
- Problems
  - §9.1 - 1, 2
  - §9.2 - 3

Chapter 9: Medians and Order Statistics

- **Chapter 9.1: Minimum and Maximum**
- Chapter 9.2: Selection in Expected Linear Time
- Chapter 9.3: Selection in Worst-Case Linear Time

Chapter 9: Medians and Order Statistics

- The ith **order statistic** of a set of n elements is the ith smallest element.
  - $i = 1$: the minimum
  - $i = n$: the maximum
- A **median** is the "halfway point" of the set:
  - If n is odd: median at $i = (n+1)/2$
  - If n is even: medians at $i = n/2$ and $i = n/2 + 1$
  - So in general: medians occur at

$$i = \left\lfloor \frac{n+1}{2} \right\rfloor \quad \text{and} \quad i = \left\lceil \frac{n+1}{2} \right\rceil$$

  - For simplicity, we refer to the **lower median**

Selection Problem Definition

Goal: Select the ith order statistic from a set of n distinct elements.

Input:

- A set A of n distinct numbers
- An integer i, where $1 \leq i \leq n$

Output:

The element $x \in A$ such that exactly $i-1$ elements are smaller than x

Although we assume distinct values, most algorithms extend to inputs with duplicates.

Baseline and Chapter Roadmap

Baseline solution:

- Sort A in $O(n \log n)$ time (e.g., using heapsort or merge sort)
- Return the ith element in the sorted array

This chapter presents faster algorithms:

- **Section 9.1:** Find the minimum and maximum efficiently
- **Section 9.2:** Randomized selection — $O(n)$ expected time
- **Section 9.3:** Deterministic selection — $O(n)$ worst-case time

Finding the Minimum in a Set of n Elements

```
MINIMUM(A, n)
min = A[1]
for i = 2 to n
    if min > A[i]
        min = A[i]
return min
```

- How many comparisons are necessary to determine the minimum of n elements?
- A natural algorithm:
  - Examine each element one at a time
  - Keep track of the smallest element seen so far

This algorithm performs $n - 1$ comparisons.

Why is $n - 1$ Comparisons Optimal?

- It's no more difficult to find the maximum — same method, same bound.
- Is this the best we can do for finding the minimum?
- **Yes.** There is a lower bound of $n - 1$ comparisons.

Tournament analogy:

- Think of each comparison as a **match**.
- The smaller of two elements "wins" each match.
- To find the overall minimum, every other element must lose at least once.
- Therefore, there must be at least $n - 1$ matches to find the winner.

Conclusion: The algorithm `MINIMUM` is **optimal** in terms of the number of comparisons.

Simultaneous Minimum and Maximum

- Some applications require computing both the minimum and the maximum of a set of n elements.
- **Example:** A graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen.
- To do so, the program must determine:
  - Minimum and maximum of all x-coordinates
  - Minimum and maximum of all y-coordinates

Naïve Solution and Optimization Goal

Naïve approach:

- Find the minimum and maximum separately
- Each takes $n - 1$ comparisons
- Total: $2n - 2 = \Theta(n)$ comparisons

Goal: Improve the leading constant while still using $\Theta(n)$ time

Optimizing with Pairwise Comparison

We can reduce the number of comparisons by processing elements in **pairs**.

- Compare each pair (a, b) with each other first
- Compare:
  - the smaller to the current minimum
  - the larger to the current maximum
- Each pair costs 3 comparisons instead of 4

Total: 3 comparisons for every 2 elements

Initialization Based on Even or Odd n

If n is odd:

- Set both min and max to the first element
- Process the remaining $n - 1$ elements in pairs

If n is even:

- Make 1 comparison between the first 2 elements
- Use the smaller as initial min, and the larger as initial max
- Process the remaining $n - 2$ elements in pairs

Total Number of Comparisons

If n is odd:

- Process $n-1$ elements in $\lfloor n/2 \rfloor$ pairs
- Total comparisons: $3\left\lfloor \frac{n}{2} \right\rfloor$

If n is even:

- 1 comparison to initialize min and max
- $n-2$ elements $\rightarrow (n-2)/2$ pairs
- Total comparisons:

$$1 + 3 \cdot \frac{n-2}{2} = \frac{3n}{2} - 2$$

In both cases:

$$\boxed{\text{At most } 3\left\lfloor \frac{n}{2} \right\rfloor \text{ comparisons}}$$

Chapter 9: Medians and Order Statistics

- Chapter 9.1: Minimum and Maximum
- **Chapter 9.2: Selection in Expected Linear Time**
- Chapter 9.3: Selection in Worst-Case Linear Time

9.2 Selection in Expected Linear Time

- The general selection problem — finding the ith order statistic for any i — may seem more difficult than simply finding the minimum.
- Yet, **surprisingly**, both have the same asymptotic running time:

$$\boxed{\Theta(n)}$$

- This section presents a **divide-and-conquer algorithm** for selection: `RANDOMIZED-SELECT`
- The algorithm builds on the structure of `QUICKSORT` (Chapter 7)

How RANDOMIZED-SELECT Differs from Quicksort

- Like `QUICKSORT`, `RANDOMIZED-SELECT` uses `RANDOMIZED-PARTITION` to divide the input array.
- **Key difference:**
  - Quicksort recursively processes **both** sides of the partition.
  - RANDOMIZED-SELECT recursively processes **only one side**.
- This difference affects the running time:

$$\texttt{Quicksort} \rightarrow \Theta(n \log n) \quad \text{(expected)}$$

$$\texttt{Randomized-Select} \rightarrow \boxed{\Theta(n)} \quad \text{(expected)}$$

RANDOMIZED-SELECT: Recursive Structure (1)

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]      // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6      return A[q]      // the pivot value is the answer
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- Line 1 checks for the base case: when subarray $A[p \dots r]$ contains only one element.
- Otherwise, line 3 calls `RANDOMIZED-PARTITION`, which splits $A[p \dots r]$ into two subarrays:

$$A[p \dots q-1] \quad \text{and} \quad A[q+1 \dots r]$$

where $A[q]$ is the pivot.

RANDOMIZED-SELECT: Recursive Structure (2)

- Elements in the left subarray are $\leq A[q]$, and those in the right are $> A[q]$.
- Line 4 computes the rank $k = q - p + 1$, the number of elements in the left (including the pivot).
- Line 5 checks:

$$\text{If } i = k \Rightarrow \text{return } A[q] \text{ as the } i\text{th smallest element}$$

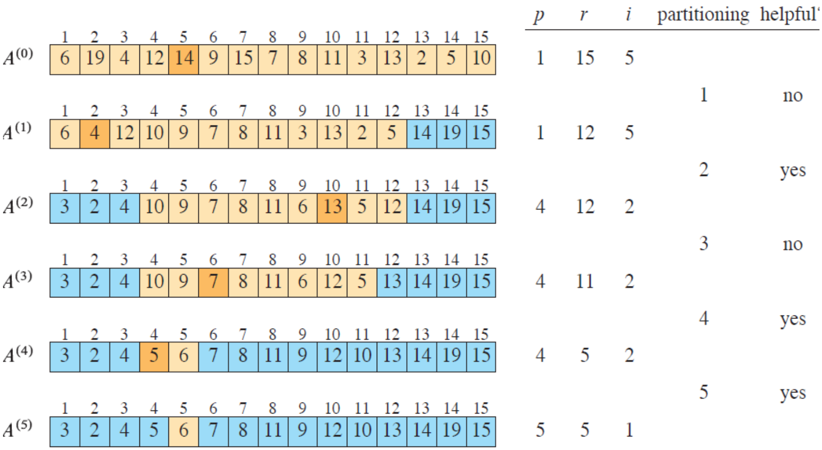
- Line 8: If $i < k$, recurse into the left subarray:

$$A[p \ldots q-1],\ \ i \text{ remains the same}$$

- Line 9: If $i > k$, recurse into the right subarray:

$$A[q+1 \ldots r],\ \text{with new rank } i - k$$

Diagram of Tracing RANDOMIZED-SELECT



Worst-Case Recurrence Relation

- In the worst case, the recursive call reduces the problem by only 1 element at a time:

$$T(n) = T(n-1) + \Theta(n)$$

- This recurrence is identical to that of `QUICKSORT` in its worst case.
- Solution to the recurrence:

$$\boxed{T(n) = \Theta(n^2)}$$

- Fortunately, since `RANDOMIZED-SELECT` is randomized, no specific input always triggers the worst-case.

Expected Time Intuition: Middle Half Pivot

- Suppose each pivot randomly selected lies in the **middle half** of the array:
  - Between the 2nd and 3rd quartiles (25th–75th percentiles)
- If the *i*th smallest element is less than the pivot:
  - All elements greater than the pivot are eliminated
  - At least the upper quartile is ignored
- If the *i*th element is greater than the pivot:
  - All elements less than the pivot are ignored
  - At least the lower quartile is ignored

At Least 1/4 of Elements are Ignored

- Either way, at least $\frac{1}{4}$ of the elements are eliminated
- So at most $\frac{3n}{4}$ elements remain in play
- The recurrence becomes:

$$T(n) = T(3n/4) + \Theta(n)$$

- By **Case 3 of the Master Theorem**, this solves to:

$$\boxed{T(n) = \Theta(n)}$$

But the Pivot is Not Always Helpful

- The pivot doesn't always fall in the middle half
- Since pivot is random, the chance it lands in the middle half is:

$$\boxed{p = \frac{1}{2}}$$

- This is modeled as a **Bernoulli trial** (success = "middle-half pivot")
- The number of trials until success follows a **geometric distribution**

$$\text{Expected trials} = \frac{1}{p} = 2$$

Final Expectation Argument

- On average:
  - Half the time, the pivot is good and reduces the problem by $\geq 1/4$
  - Half the time, it may not help as much
- But good pivots dominate the cost:
  - They eliminate a significant portion of the array
- Therefore, the **expected running time** of `RANDOMIZED-SELECT` is:

$$\boxed{\mathbb{E}[T(n)] = \Theta(n)}$$

Formal Analysis - Tracking Elements: Sets $A^{(j)}$

- Let $A^{(j)}$ denote the set of elements still in play after the jth partitioning.
- So:

$$A^{(0)} = A \quad \text{(initial full array)}$$

- After each call to `RANDOMIZED-SELECT`, the pivot is removed from play:

$$|A^{(0)}| > |A^{(1)}| > |A^{(2)}| > \cdots$$

- For convenience, we treat $A^{(0)}$ as the original input array.

What is a "Helpful" Partitioning?

- We call the jth partitioning **helpful** if:

$$|A^{(j)}| \leq \frac{3}{4}|A^{(j-1)}|$$

- This means that at least $\frac{1}{4}$ of the current elements are eliminated from further consideration.
- If the pivot falls into the **middle half**, the partition is helpful:
  - Because either the lower or upper quartile gets discarded
- A helpful partitioning corresponds to a **successful Bernoulli trial**.

Lemma 9.1 — Probability of Helpful Partitioning

Claim: A partitioning is helpful with probability at least $\frac{1}{2}$

Proof Sketch:

- Define the **middle half** of the subarray:
  - All but the smallest $\lfloor n/4 \rfloor - 1$ and largest $\lfloor n/4 \rfloor - 1$ elements
- If the pivot lands in this middle half:
  - At least $\lfloor n/4 \rfloor$ elements are eliminated
  - Remaining elements:

$$\leq n - \lfloor n/4 \rfloor = \lfloor 3n/4 \rfloor$$

$\rightarrow$ partition is helpful

Proof (continued): Probability Bound

Goal: Show that pivot lands in middle half with probability $\geq \frac{1}{2}$

- Total size of non-middle elements:

$$2(\lfloor n/4 \rfloor - 1)$$

- So, probability that pivot is *not* in the middle half:

$$\leq \frac{2(\lfloor n/4 \rfloor - 1)}{n} \leq \frac{n/2}{n} = \frac{1}{2}$$

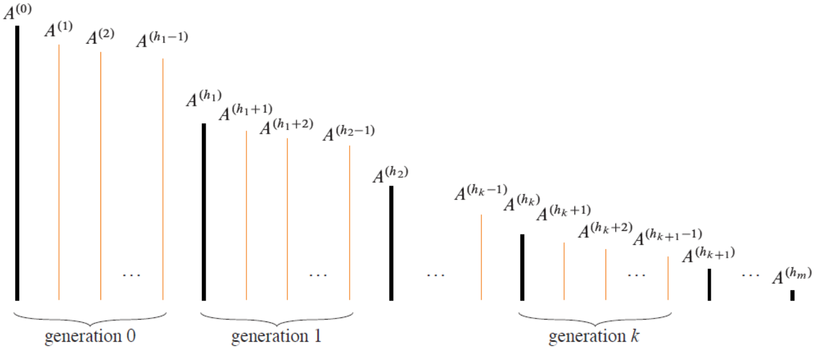
- Therefore, probability that pivot **is** in middle half:

$$\geq 1 - \frac{1}{2} = \boxed{\frac{1}{2}}$$

Conclusion: A partitioning is helpful with probability at least $\frac{1}{2}$

Expected Time of RANDOMIZED-SELECT

Theorem 9.2. `RANDOMIZED-SELECT` on an input array of n distinct elements has an expected running time of $\Theta(n)$.



Proof Overview

- We group recursive partitioning steps into "generations."
- Define the sequence $\langle h_0, h_1, \ldots, h_m \rangle$, where each h_k is the index of a "helpful" partitioning that shrinks the remaining input by at least $1/4$.
- Let $A^{(j)}$ denote the subarray still in play after the jth partitioning. Initially, $A^{(0)} = A$, the entire array.
- Define $n_k = |A^{(h_k)}|$, the size of the subarray at the beginning of generation k. Then:

$$n_k \leq \left(\frac{3}{4}\right)^k n$$

Generations and Random Variables

- Generation k: the steps between h_k and $h_{k+1} - 1$.
- Let $X_k = h_{k+1} - h_k$, i.e., number of partitioning steps in generation k.
- From Lemma 9.1: each partitioning has at least a $1/2$ probability of being helpful (success in a Bernoulli trial).
- So expected length of each generation:

$$\mathbb{E}[X_k] \leq 2$$

- Each subarray processed in generation k has size $\leq n_k \leq \left(\frac{3}{4}\right)^k n$

Bounding the Number of Comparisons

- Each partitioning compares the pivot to every other element in its subarray
- For generation k: X_k partitionings, each on size $\leq \left(\frac{3}{4}\right)^k n$
- So total comparisons in generation k:

$$X_k \cdot \left(\frac{3}{4}\right)^k n$$

- Total comparisons across all generations:

$$\sum_{k=0}^{m-1} X_k \cdot \left(\frac{3}{4}\right)^k n$$

Expected Total Comparisons

Taking expectation on the total:

$$\begin{aligned}
\mathbb{E}[\text{Total Comparisons}] &= \sum_{k=0}^{m-1} \mathbb{E}[X_k] \cdot \left(\frac{3}{4}\right)^k n \\
&\leq 2n \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 2n \cdot \frac{1}{1 - 3/4} = 8n
\end{aligned}$$

Conclusion: The expected number of comparisons is $\mathcal{O}(n)$. Since at least $\Omega(n)$ work is done in the first partition, we conclude:

$$\boxed{\mathbb{E}[T(n)] = \Theta(n)}$$

Part III: Data Structures

- Chapter 10: Elementary Data Structures
- Chapter 11: Hash Tables
- Chapter 12: Binary Search Trees
- Chapter 13: Red-Black Trees

Introduction to Dynamic Sets

Sets are fundamental in both mathematics and computer science.

- Unlike mathematical sets (which are static), algorithmic sets can change over time.
- Such changeable sets are called **dynamic sets**.
- The next four chapters introduce techniques for representing and manipulating finite dynamic sets.

Operations on Dynamic Sets

Dynamic sets support various operations depending on algorithmic needs.

- A common type is a **dictionary**, supporting:
  - `INSERT`, `DELETE`, and `SEARCH`
- Other structures support more complex operations:
  - E.g., **min-priority queues** support `INSERT` and `EXTRACT-MIN`
- Choice of implementation depends on the operations required.

Operations on Dynamic Sets

Operations are categorized into:

- **Queries:** Retrieve information without changing the set
- **Modifications:** Alter the contents of the set

Common Operations:

- `SEARCH(S, k)`: Return pointer to element with key k, or `NIL`
- `INSERT(S, x)`: Add element x to set S
- `DELETE(S, x)`: Remove element pointed to by x from S

Learn Data Structures in Part III

Unsorted Array:

- INSERT, DELETE: $\Theta(1)$ time
- SEARCH, MINIMUM, etc.: $\Theta(n)$ time

Sorted Array:

- MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR: $\Theta(1)$ time
- SEARCH: $O(\lg n)$ via binary search
- INSERT, DELETE: $\Theta(n)$ in worst case

But, for example, **Heap:**

- INSERT/DELETE: $O(\lg n)$
- MINIMUM/MAXIMUM: $\Theta(1)$
- SEARCH, SUCCESSOR, PREDECESSOR: $\Theta(n)$ in general

Upcoming data structures (Chapter 10–13) improve these time bounds.

Chapter 10: Elementary Data Structures

- **Chapter 10.1: Simple Array-Based Structures (Arrays, Matrices, Stacks, Queues)**
- Chapter 10.2: Linked Lists
- Chapter 10.3: Representing Rooted Trees

Arrays: Memory Layout

Memory layout: Arrays are allocated as a contiguous block of memory with a fixed stride (i.e., same number of bytes per element).
Index-based access: To access $A[i]$, the system computes the address using:

$$\text{address} = a + i \cdot b$$

where:

- a: base address
- b: fixed byte size per element

Access Time and Variable Element Sizes

- Under the RAM model: access to any $A[i]$ takes $\Theta(1)$ time.
- Direct addressing requires elements to be of **fixed size**.
- If elements are **not the same size**, we cannot compute addresses using a simple formula, so direct addressing fails.
- Instead, store **pointers**:
  - Each entry stores the address of a variable-size object
  - Dereference the pointer to access the object
- Pointers themselves are fixed-size (e.g., 4 or 8 bytes)

Matrix Representation Overview

Matrix: $m \times n$ matrix M

- Represented as a 2D array using 1D arrays
- Two common linear storage schemes:
  - **Row-major:** store row-by-row
  - **Column-major:** store column-by-column

Row-Major vs Column-Major Order

Let $M[i][j]$ be the element at row i and column j.

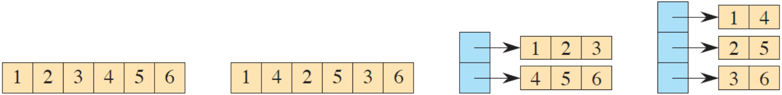
- **Row-major:** element index $= n \cdot i + j$
- **Column-major:** element index $= i + m \cdot j$

Example Matrix:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- Row-major: $[1, 2, 3, 4, 5, 6]$
- Column-major: $[1, 4, 2, 5, 3, 6]$

Multi-Array Representations



(a) (b) (c) (d)

Row Pointers:

- One array for each row
- Master array holds pointers to row arrays
- Access $M[i][j]$ via $A[i][j]$

Column Pointers:

- One array for each column
- Master array holds pointers to column arrays
- Access $M[i][j]$ via $A[j][i]$

Block Representation

Block Storage:

- Divide the matrix into blocks (e.g., 2×2 blocks)
- Store blocks contiguously in memory

Example: 4×4 **matrix stored in** 2×2 **blocks:**

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Block Order: $[1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16]$

Stacks: Overview and Intuition

Stack Basics

- Stack = dynamic set with **LIFO** (Last-In, First-Out) behavior
- `INSERT` $\rightarrow$ `PUSH`, `DELETE` $\rightarrow$ `POP`
- Analogy: Cafeteria plate dispenser
- Most recently inserted element is the first to be removed

Stacks: Array Implementation

- Use array $S[1..n]$ to hold stack elements
- Attributes:
  - $S.\texttt{top}$: index of most recently inserted element
  - $S.\texttt{size}$: capacity of the stack (i.e., n)
- Stack consists of elements $S[1..S.\texttt{top}]$

Diagram: Example with $S.\texttt{top} = 4$

Stack Procedures

```
STACK-EMPTY(S)
  if S.top == 0
      return TRUE
  else return FALSE

PUSH(S, x)
  if S.top == S.size
      error "overflow"
  else S.top = S.top + 1
      S[S.top] = x

POP(S)
  if STACK-EMPTY(S)
      error "underflow"
  else S.top = S.top - 1
      return S[S.top + 1]
```

Stack Operations and Overflow/Underflow

Special Conditions

- $S.\mathtt{top} = 0 \Rightarrow$ Stack is empty
- $S.\mathtt{top} = S.\mathtt{size} \Rightarrow$ Stack is full

Error handling:

- `POP` on empty stack: underflow error
- `PUSH` when full: overflow error

All operations run in $\Theta(1)$ time.

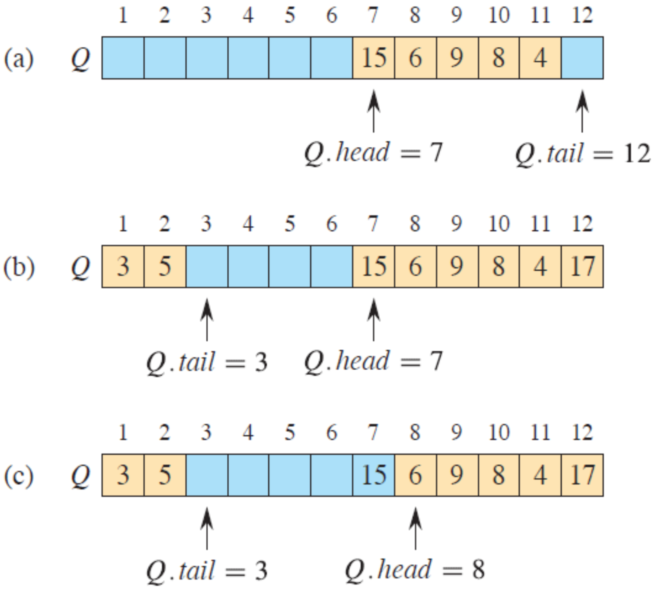
Queues: Overview

- **ENQUEUE**: Insert operation (adds to tail)
- **DEQUEUE**: Delete operation (removes from head)
- Queue follows **FIFO** (First-In, First-Out) policy
- Similar to a line of customers: new arrivals go to the back, service from the front

Queues: Array-Based Implementation

- Use array $Q[1 \dots n]$ to store queue elements
- Attributes:
  - $Q.size$: total size of the array (capacity n)
  - $Q.head$: index of the front (dequeue from here)
  - $Q.tail$: index of the next insertion (enqueue to here)
- Queue elements stored from $Q.head$ to $Q.tail - 1$
- Indices **wrap around** circularly: index 1 follows n
- We store at most $n - 1$ elements to distinguish full vs. empty:
  - **Empty queue:** $Q.head = Q.tail$
  - **Full queue:** $Q.head = Q.tail + 1$

Queues: Example



Queues: Special Conditions

- **Empty:** $Q.head = Q.tail$
- **Full:** $Q.head = Q.tail + 1$ (or $Q.head = 1$ and $Q.tail = Q.size$)
- **Underflow:** Attempt to dequeue from empty queue
- **Overflow:** Attempt to enqueue into full queue

Queue Operations

ENQUEUE(Q, x)

```
Q[Q.tail] = x
if Q.tail == Q.size
        Q.tail = 1
else Q.tail = Q.tail + 1
```

DEQUEUE(Q)

```
x = Q[Q.head]
if Q.head == Q.size
        Q.head = 1
else Q.head = Q.head + 1
return x
```

Appendix

What is a Bernoulli Trial?

- A **Bernoulli trial** is a random experiment with exactly two outcomes:
$$\text{Success} \quad \text{or} \quad \text{Failure}$$
- Each trial is independent, and the probability of success is fixed:
$$\text{Success with probability } p, \quad \text{Failure with probability } 1-p$$
- **Examples:**
  - Flipping a coin (Heads = success, Tails = failure)
  - Rolling a die and checking if you get a 6
  - Picking a random pivot and checking if it falls into the middle half

Geometric Distribution: Trials Until First Success

- The **geometric distribution** models the number of independent Bernoulli trials until the first success occurs.
- If each trial succeeds with probability p, then:

$$\mathbb{E}[\text{Number of trials until success}] = \frac{1}{p}$$

Proof:

$$\mathbb{E}[X] = \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1} \cdot p$$

$$= p \sum_{k=1}^{\infty} k(1-p)^{k-1}$$

Let $q = 1 - p$. Use the identity:

$$\sum_{k=1}^{\infty} kq^{k-1} = \frac{1}{(1-q)^2} = \frac{1}{p^2}$$

Therefore: $\mathbb{E}[X] = p \cdot \frac{1}{p^2} = \frac{1}{p}$

Question?