Chapter 5. Probabilistic Analysis and Randomized Algorithms

Joon Soo Yoo

April 1, 2025

Chapter 5: Probabilistic Analysis and Randomized Algorithms

- Chapter 5.1: The hiring problem
- Chapter 5.2: Indicator random variables
- Chapter 5.3: Randomized algorithms

Assignment

- Read §5.1, §5.2, §5.3
- Problems:
  - §5.2 - #2, 5, 6
  - §5.3 - #2, 4

Chapter 5: Probabilistic Analysis and Randomized Algorithms

- **Chapter 5.1: The hiring problem**
- Chapter 5.2: Indicator random variables
- Chapter 5.3: Randomized algorithms

5.1 The Hiring Problem – Scenario

- You need to hire a new office assistant.
- An employment agency sends you one candidate per day.
- For each candidate:
  - You pay a small fee to interview.
  - If you hire, there is a larger cost (firing the current assistant and hiring the new one).
- **Goal:** Always have the best candidate (highest quality) so far.

5.1 The Hiring Problem – Strategy

- Interview candidate i.
- If candidate i is better than the current assistant:
  - Fire the current assistant.
  - Hire candidate i.
- We wish to estimate the cost incurred by this strategy.

```
HIRE-ASSISTANT(n)
best = 0            // candidate 0 is a least-qualified dummy candidate
for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
        best = i
        hire candidate i
```

Cost Model

- Each interview costs c_i (small cost).
- Each hiring costs c_h (large cost).
- Let m be the number of hires.
- **Total cost:**

$$O(c_i \cdot n + c_h \cdot m)$$

- Since you always interview n candidates, the focus is on analyzing $c_h \cdot m$.

A Common Computational Paradigm

- The hiring problem models the process of **tracking a maximum** (or minimum).
- Many algorithms work by processing elements in sequence and updating a current "winner".
- Similar ideas appear in:
  - Greedy algorithms
  - Online algorithms
  - Real-time decision making

Worst-Case Analysis

- In the worst case, you hire **every** candidate.
- This occurs when candidates arrive in **increasing order of quality**.
- Then, the total hiring cost is:

$$O(c_h \cdot n)$$

- However, this extreme case is rare.

Why Probabilistic Analysis?

- In many real-world scenarios, the order in which candidates arrive is unpredictable.
- We might assume that candidates arrive in a **random order**.
- Then, the question becomes:

  What is the expected number of hires?

- This leads us to use **probabilistic analysis**.

What is Probabilistic Analysis?

- It is the use of probability to analyze algorithms.
- Typically applied when:
  - The input is drawn from a random distribution.
  - We average the cost (or running time) over all possible inputs.
- In the hiring problem, we assume the order (or ranks) of candidates is random.
- The result of such analysis is called the **average-case performance**.

Random Order Model for Hiring

- Each candidate is assigned a unique rank from 1 to n (with higher rank meaning better).
- Let $\texttt{rank}(i)$ denote the rank of candidate i.
- Saying the candidates arrive in a **random order** means:
  - The list $(\texttt{rank}(1), \texttt{rank}(2), \ldots, \texttt{rank}(n))$ is a uniform random permutation.
  - There are $n!$ equally likely orderings.

Hiring Problem – Controlled Randomness

- In the original model, you hope the candidates arrive in random order.
- To be sure, the agency sends you the complete list of n candidates.
- Then, each day you **randomly choose** a candidate to interview.
- This way, you **enforce** a random order.

What is a Randomized Algorithm?

- A randomized algorithm uses a random-number generator to make decisions.
- Its behavior depends on:
  - The input, and
  - Random choices made during execution.
- Even on a fixed input, different runs may yield different behaviors.

Random-Number Generator: `RANDOM(a, b)`

- `RANDOM(a, b)` returns an integer uniformly at random in the interval $[a, b]$.
- Examples:
  - `RANDOM(0, 1)` returns 0 or 1 (each with probability $1/2$).
  - `RANDOM(3, 7)` returns 3, 4, 5, 6, or 7 (each with probability $1/5$).
- Each call is independent of previous calls.
- Think of it as rolling a $(b - a + 1)$-sided die.

Summary of Key Differences

- **Average-case Analysis:**
  - Randomness comes from the **input** (e.g., a uniform random permutation).
  - The algorithm is deterministic.
- **Expected Running Time:**
  - Randomness is introduced by the **algorithm**'s internal choices.
  - Even on a fixed input, outcomes vary due to random decisions.

Probability Review: Appendix C.2

- Probability is fundamental in analyzing **randomized algorithms**.
- Used to describe and reason about outcomes in uncertain environments.
- Examples:
  - Flipping coins
  - Randomized quicksort
  - Hashing collisions

Sample Space and Events

- A **sample space** S is the set of all possible outcomes.
- An **outcome** is a single result of an experiment.
- An **event** is a subset of S.
- Example: Flipping 2 distinguishable coins
  - $S = \{HH, HT, TH, TT\}$
  - Event: "One head, one tail" $= \{HT, TH\}$
  - Event: Certain event $= S$, Null event $= \emptyset$

Probability Axioms

Let $\Pr\{A\}$ denote the probability of event A.

1. Non-negativity: $\Pr\{A\} \geq 0$
2. Normalization: $\Pr\{S\} = 1$
3. Additivity: For **mutually exclusive** events A and B, $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$

Example: In uniform distribution over $S = \{HH, HT, TH, TT\}$, each outcome has $\Pr = 1/4$.

Useful Probability Rules

- $\Pr\{\emptyset\} = 0$
- If $A \subseteq B$, then $\Pr\{A\} \leq \Pr\{B\}$
- Complement Rule:

$$\Pr\{A^c\} = 1 - \Pr\{A\}$$

- General Addition Rule:

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\}$$

Discrete Probability Distributions

- Sample space S is **finite** or **countably infinite**
- For any event A:

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\}$$

- If uniform: all outcomes equally likely.
- Example: n fair coin flips
  - $S = \{H, T\}^n$, $|S| = 2^n$
  - Each string occurs with $\Pr = 1/2^n$
  - Event: exactly k heads has probability:

$$\binom{n}{k} / 2^n$$

Continuous Uniform Distribution

- Sample space is interval $[a, b]$ on real line
- **Equal probability density** throughout the interval
- Probability of event $[c, d]$:

$$\Pr\{[c, d]\} = \frac{d - c}{b - a}$$

- Probability of any **single point** is zero
- Events are intervals (open, closed, finite unions, etc.)

Conditional Probability

Definition: Given events A and B with $\Pr\{B\} > 0$,

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}}$$

- Read as: "Probability of A given B occurred"
- Narrows sample space to event B

Example: Two coin flips. Given at least one head, what's the probability of two heads?

$$\Pr\{HH \mid HH, HT, TH\} = 1/3$$

Independence

Two events A and B are **independent** if:

$$\Pr\{A \cap B\} = \Pr\{A\} \cdot \Pr\{B\}$$

Or equivalently:

$$\Pr\{A \mid B\} = \Pr\{A\} \text{ (if } \Pr\{B\} \neq 0)$$

Examples:

- Independent: Two fair coin flips
- Not independent: Welded coins (both always same)

Pairwise vs Mutual Independence

- **Pairwise Independent:** Every pair A_i, A_j satisfies:

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\} \cdot \Pr\{A_j\}$$

- **Mutually Independent:** For every subset $\{A_{i_1}, ..., A_{i_k}\}$,

$$\Pr\{A_{i_1} \cap \cdots \cap A_{i_k}\} = \prod_{j=1}^{k} \Pr\{A_{i_j}\}$$

- **Important:** Pairwise independence does *not* imply mutual independence

Chapter 5: Probabilistic Analysis and Randomized Algorithms

- Chapter 5.1: The hiring problem
- **Chapter 5.2: Indicator random variables**
- Chapter 5.3: Randomized algorithms

5.2 Indicator Random Variables

- Used to connect **probabilities** to **expectations**.
- Useful in analyzing repeated random trials in algorithms.
- Given an event A in a sample space S, define:

$$I\{A\} = \begin{cases} 1 & \text{if event } A \text{ occurs} \\ 0 & \text{otherwise} \end{cases}$$

Example: One Coin Flip

- Flip a fair coin: $S = \{H, T\}$ with $P(H) = P(T) = 1/2$
- Let $X_H = I\{H\}$ be the indicator for "heads"
- Then:
$$\mathbb{E}[X_H] = 1 \cdot P(H) + 0 \cdot P(T) = \frac{1}{2}$$
- So the expected number of heads from one fair flip is $\boxed{1/2}$

Lemma 5.1 – Expectation of an Indicator

Lemma: For any event A, and indicator $X_A = I\{A\}$:

$$\mathbb{E}[X_A] = P(A)$$

Proof:

$$\mathbb{E}[X_A] = 1 \cdot P(A) + 0 \cdot (1 - P(A)) = P(A)$$

- This is the core reason why indicators are useful in expectation analysis!

Multiple Coin Flips

- Suppose we flip a fair coin n times.
- Let $X_i = I\{\text{flip } i \text{ is heads}\}$
- Define $X = \sum_{i=1}^{n} X_i$ = total number of heads

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right]$$

Linearity of Expectation

- Linearity of expectation says:

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

- Even if the X_i's are not independent!
- Since $\mathbb{E}[X_i] = \frac{1}{2}$ for each i:

$$\mathbb{E}[X] = \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$$

- So the expected number of heads in n fair flips is $\boxed{n/2}$

Analyzing Hiring with Indicator Variables

- Let X = total number of hires.
- Instead of computing $\mathbb{E}[X]$ directly, break it down using indicators.
- Define:

$$X_i = I\{\text{candidate } i \text{ is hired}\} \in \{0, 1\}$$

- Then:

$$X = \sum_{i=1}^{n} X_i$$

Expected Number of Hires

- Let X_i be the indicator variable for the event: *"Candidate i is hired."* So:

$$X_i = \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{otherwise} \end{cases}$$

- Total number of hires is:

$$X = \sum_{i=1}^{n} X_i$$

- By linearity of expectation:

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

- Candidate i is hired iff they are better than all previous $i-1$ candidates.
- Since the input order is random, each of the first i candidates is equally likely to be the best so far:

$$\Pr(\text{candidate } i \text{ is best among } 1 \text{ to } i) = \frac{1}{i}$$

- Therefore:

$$\mathbb{E}[X_i] = \Pr(\text{candidate } i \text{ is hired}) = \frac{1}{i}$$

Expected Number of Hires (continued)

- Now compute the total expectation:

$$\mathbb{E}[X] = \sum_{i=1}^{n} \frac{1}{i}$$

- This is the harmonic series:

$$H_n = \ln n + \gamma + \varepsilon_n$$

  where $\gamma \approx 0.5772$ is the **Euler–Mascheroni constant**, and $\varepsilon_n \to 0$

- In algorithm analysis, we simplify this as:

$$\boxed{\mathbb{E}[X] = \ln n + O(1)}$$

- On average, you hire only about $\ln n$ people, even though you interview n!

Hiring Cost – Summary Result

- Recall total hiring cost is:

$$\text{Cost} = c_i \cdot n + c_h \cdot X$$

- Average-case number of hires:

$$\mathbb{E}[X] = \ln n + O(1)$$

- So, average-case hiring cost:

$$\boxed{O(c_i \cdot n + c_h \cdot \ln n)}$$

- Much better than worst-case cost of $O(c_h \cdot n)$!

Chapter 5: Probabilistic Analysis and Randomized Algorithms

- Chapter 5.1: The hiring problem
- Chapter 5.2: Indicator random variables
- **Chapter 5.3: Randomized algorithms**

5.3 Randomized Algorithms – Motivation

- What if we don't know the distribution of inputs?
- Then we can't rely on average-case analysis.
- Instead, we can design a **randomized algorithm**.
- Idea: enforce randomness in the algorithm, not in the input.
- Example: shuffle the input before processing.

Hiring Problem – Input Sensitivity

- Deterministic algorithm, random input:
  - Input $A_1 = [1, 2, 3, ..., 10] \to$ 10 hires (worst case)
  - Input $A_2 = [10, 9, ..., 1] \to$ 1 hire (best case)
  - Input $A_3 = [5, 2, 1, 8, 4, 7, 10, 9, 3, 6] \to$ 3 hires
- The cost depends heavily on input order.

Randomized Algorithm Approach

- Suppose input is fixed: $A_3 = [5, 2, 1, 8, ...]$
- Randomized algorithm shuffles the input first.
- Each execution produces a different order:
  - Shuffle 1 $\rightarrow A_1 \rightarrow$ 10 hires
  - Shuffle 2 $\rightarrow A_2 \rightarrow$ 1 hire
  - Shuffle 3 $\rightarrow$ something in between
- Randomness is now inside the algorithm, not the input.

Key Difference

- **Probabilistic Analysis**:
  - Assumes input is random
  - Algorithm is deterministic
  - Cost depends on input distribution
- **Randomized Algorithm**:
  - Input is arbitrary (even adversarial)
  - Algorithm uses randomness (e.g., shuffle)
  - Cost is averaged over algorithm's own random choices

Why Randomized Algorithms Are Powerful

- Deterministic algorithm:
  - Worst-case input can force bad performance
- Randomized algorithm:
  - No input can reliably trigger worst-case
- Randomness protects you from malicious or unlucky inputs!

RANDOMIZED-HIRE-ASSISTANT Procedure

Procedure:

1. Randomly permute the list of n candidates
2. Run `HIRE-ASSISTANT(n)` on the permuted list

Why it works:

- The random permutation simulates the same setting as the average-case analysis.
- But now it works for **any input**, even adversarial ones!
- The expected number of hires is still:

$$\boxed{O(\ln n)}$$

Probabilistic vs Randomized: Lemma 5.2 vs 5.3

- **Lemma 5.2 (Average-case Analysis):**
  - Assumes input is a uniform random permutation
  - Algorithm is deterministic
  - Expected cost: $O(c_h \cdot \ln n)$
- **Lemma 5.3 (Randomized Algorithm):**
  - Input is arbitrary (even adversarial)
  - Algorithm shuffles input (randomized)
  - Expected cost: $O(c_h \cdot \ln n)$ — same result!
- **Key difference:** Randomization is in the input (5.2) vs. in the algorithm (5.3)

Procedure: `RANDOMLY-PERMUTE(A)`

RANDOMLY-PERMUTE(A)

for $i = 1$ to n **do**
  swap $A[i]$ with $A[\texttt{Random}(i, n)]$
end for

Uniform Permutation: High-Level Goal

- We want to produce a **uniform random permutation** of n elements.
- There are $n!$ possible permutations.
- Goal: Each should occur with probability $1/n!$.
- Procedure: `RANDOMLY-PERMUTE(A)`
  - For $i = 1$ to n:
  - Swap $A[i]$ with $A[\mathtt{RANDOM}(i, n)]$
- How do we prove this generates a uniform distribution?
- **We use a loop invariant.**

Loop Invariant: Key Idea

Invariant: Just before iteration i, the subarray $A[1 \ldots i-1]$ contains each $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.

We prove:

- Initialization (base case): Invariant holds before iteration 1.
- Maintenance: Invariant holds after iteration i.
- Termination: When loop ends, $A[1..n]$ is a uniform permutation.

Initialization (Base Case)

- Before iteration $i = 1$, subarray $A[1..0]$ is empty.
- A 0-permutation has only one possibility: the empty sequence.
- So it appears with probability 1.
- Invariant holds: $A[1..0]$ is trivially uniform.

Maintenance: Extending to i Elements

- Assume invariant holds before iteration i.
- Subarray $A[1..i-1]$ is uniform $(i-1)$-permutation.
- Let $x_1, x_2, \ldots, x_{i-1}$ be elements in $A[1..i-1]$.
- Let x_i be selected from $A[i..n]$ and placed in $A[i]$.

Let:

- E_1: event that $A[1..i-1] = (x_1, \ldots, x_{i-1})$.
- E_2: event that x_i is placed into $A[i]$.

Then: $\Pr[E_1 \cap E_2] = \Pr[E_1] \cdot \Pr[E_2 \mid E_1]$

Maintenance: Probability Calculation

- From invariant: $\Pr[E_1] = \frac{(n-i+1)!}{n!}$
- $\Pr[E_2 \mid E_1] = \frac{1}{n-i+1}$ (uniform choice from $A[i..n]$)
- So:

$$\Pr[E_1 \cap E_2] = \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} = \frac{(n-i)!}{n!}$$

- This holds for every i-permutation.
- So the invariant holds after iteration i.

Termination and Conclusion

- Loop ends after n iterations.
- At $i = n + 1$, $A[1..n]$ is a full n-permutation.
- Invariant implies each permutation occurs with:

$$\frac{(n-n)!}{n!} = \frac{1}{n!}$$

- **Conclusion:** RANDOMLY-PERMUTE produces a uniform random permutation in $O(n)$ time, in-place.

CLRS 5.3-3: Does PERMUTE-WITH-ALL Produce a Uniform Permutation?

Algorithm:

PERMUTE-WITH-ALL

```
for i = 1 to n do
    swap A[i] with A[RANDOM(1, n)]
end for
```

Question: Does this algorithm produce a *uniform random permutation*? That is, does it generate each of the $n!$ permutations with probability exactly $1/n!$?

Proof: Loop Invariant Setup for PERMUTE-WITH-ALL

Define Loop Invariant: Just before iteration i, the subarray $A[1 \mathinner{..} i-1]$ contains a **uniform random** $(i-1)$-permutation of the input. That is, each such permutation appears with probability:

$$\frac{(n-i+1)!}{n!}$$

Initialization (i = 1):

- Subarray $A[1 \mathinner{..} 0]$ is empty.
- Only one 0-permutation (the empty sequence), so appears with probability 1.
- This matches $\frac{n!}{n!} = 1$

Invariant holds at initialization.

Termination (i = n + 1):

- Subarray $A[1 \mathinner{..} n]$ must be a full n-permutation.
- The invariant would imply that each permutation occurs with probability $\frac{(n-n)!}{n!} = \frac{1}{n!}$.
- This is the desired result — **if** maintenance holds.

Proof: Maintenance Fails for `PERMUTE-WITH-ALL`

Maintenance Goal: Assume the invariant holds before iteration i (event E_1):

$$\Pr[E_1] = \frac{(n-i+1)!}{n!}$$

We want to extend $A[1 \mathinner{..} i-1]$ to a uniform i-permutation. Let E_2 be the event that $A[i]$ is correctly chosen.

In `PERMUTE-WITH-ALL`, we choose from the full array:

$$\Pr[E_2 \mid E_1] = \frac{1}{n} \Rightarrow \Pr[E_1 \cap E_2] = \frac{(n-i+1)!}{n!} \cdot \frac{1}{n} \neq \frac{(n-i)!}{n!}$$

Conclusion: Maintenance step fails — the probability of constructing each i-permutation is incorrect. $\Rightarrow$ The loop invariant does not hold, and `PERMUTE-WITH-ALL` is **not uniform**.

Part II: Sorting and Order Statistics

- Chapter 6: Heapsort
- Chapter 7: Quicksort
- Chapter 8: Sorting in linear time
- Chapter 9: Medians and order statistics

Part II: Sorting and Order Statistics

What's in this part?

- Fundamental sorting algorithms: insertion sort, merge sort, heapsort, quicksort
- Comparison-based sorting and its limitations
- Non-comparison sorting: counting sort, radix sort, bucket sort
- Selecting the ith smallest element — order statistics

Big Picture: Sorting is essential in both theory and practice — efficient, optimal, and widely applicable.

Sorting — The Problem and Structure

Goal: Rearrange n input elements so that:

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

- We often sort **records** (e.g., students, orders) by a **key** (e.g., score).
- Satellite data (e.g., name, address) must move with the key.
- Efficient sorting may use **pointers** instead of physically moving records.

Four Core Sorting Algorithms

- **Insertion Sort:** Simple, quadratic time, but fast for small inputs. In-place.
- **Merge Sort:** $O(n \log n)$ time, divide-and-conquer, not in-place.
- **Heapsort:** $O(n \log n)$, in-place, uses heaps (priority queues).
- **Quicksort:** Fast in practice, $O(n \log n)$ expected time, $O(n^2)$ worst-case.

All four are **comparison sorts**.

Lower Bound for Comparison Sorting

- **Decision Tree Model:** Compares elements to determine order.
- **Key Result:**

  Any comparison-based sort requires at least $\Omega(n \log n)$ comparisons (worst-case).

- Merge sort and heapsort are **asymptotically optimal**.

Non-Comparison Sorts (Faster When Possible)

To go faster than $\Omega(n \log n)$, we must **avoid comparisons**.

- **Counting Sort:** When keys are integers in $[0, k]$. Time: $\Theta(n + k)$
- **Radix Sort:** For digit-based numbers. Time: $\Theta(d(n + k))$
- **Bucket Sort:** For uniform real numbers in $[0, 1)$. Time: $\Theta(n)$ average-case

Assumption: Input keys must satisfy special structure or distribution.

Order Statistics (Chapter 9)

- ***i*th order statistic:** The ith smallest element in an array
- Can solve with sorting: $O(n \log n)$
- **Randomized Select:** $O(n)$ expected time
- **Deterministic Select:** $O(n)$ worst-case time (but more complex)

Use case: Find the median, minimum, maximum, etc. without full sort.

Question?