Lecture Notes: Randomization in Algorithms

Lechière

August 27, 2025

1 Introduction

Randomization in algorithms is quite common these days, so a long introduction is not needed.

Definition 1. A **randomized algorithm** is one that has access to a true random number or random bit generator and can use the numbers/bits to make decisions.

Historically, people have used random sampling in surveys and data collection. Most people credit the Metropolis-Hastings algorithm for numerical integration and sampling as the first non-trivial use of randomness in algorithms. It is an early example of a situation where randomness was not only powerful but also, in a sense, the only way. In modern computing, randomized algorithms are ubiquitous.

The goal of this course is to expose you to randomized algorithms from a theoretical computer science perspective. We will cover:

- Some well-known algorithms
- Tools to help design and analyze randomized algorithms
- Basic notions of complexity related to randomization in computing
- Application to a few areas

Algorithms can be quite complicated to design and analyze, and the addition of randomization makes it even more so. For this reason, one has to learn certain "templates." Thus, there will be a bit of emphasis on tools.

1.1 Today's Tool: Linearity of Expectation

Suppose $X = \sum_{i=1}^{n} a_i X_i$, where $X_1, X_2, \ldots, X_n$ are random variables over some probability space (Ω, Pr) with finite expectation.

Then, by the linearity of expectation:

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

A key advantage is that the random variables X_i can be dependent.

2 Randomized Quicksort

Quicksort was developed by Hoare in 1959. It works well in practice since it is in-place, does not require additional memory, and has good cache behavior if implemented properly. The deterministic version can run in $O(n^2)$ time in the worst case (e.g., on an already sorted array).

We will analyze Randomized Quicksort. Assume the input array A has distinct elements.

Algorithm: Quicksort(A[1..n])

1. If n=1, return A.

2. Pick a pivot element. In Randomized Quicksort, the pivot is $A[i]$ where i is chosen uniformly at random from $\{1, 2, \ldots, n\}$.

3. Use the pivot to split $A[1..n]$ into two subarrays, one with elements less than the pivot and one with elements greater than the pivot. This step takes $O(n)$ time.

4. Recursively sort the two subarrays.

Theorem 1. *Randomized Quicksort runs in an expected time of $O(n \log n)$ on an array of n elements.*

We will also later show that it runs in $O(n \log n)$ time "with high probability."

2.1 Analysis of Expected Running Time

2.1.1 Recursion-Based Analysis

Let $T(n)$ be the expected number of comparisons that Randomized Quicksort makes on an array of n elements. The recurrence relation is:

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

with the base case $T(1) = 0$. **Exercise:** Show that $T(n) = O(n \log n)$. This recurrence is easy to write down, but the analysis is not very intuitive or insightful.

2.1.2 A Slick Analysis using Indicator Variables

Let the input array be $A = [a_1, a_2, \ldots, a_n]$. Let the sorted version of the array be $A' = [a'_1, a'_2, \ldots, a'_n]$.

- Let X be the total number of comparisons.
- Let X_{ij} be the indicator random variable for the event E_{ij} that the algorithm compares a'_i and a'_j.

The total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$. By linearity of expectation:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[E_{ij}]$$

Lemma 1. $Pr[E_{ij}] = \frac{2}{j-i+1}$.

Informal Proof. The elements a'_i and a'_j are compared if and only if one of them is chosen as a pivot from the set $\{a'_i, a'_{i+1}, \ldots, a'_j\}$ before any other element in that set is chosen as a pivot. Since any element in this set is equally likely to be the first one chosen as a pivot, the probability that this is either a'_i or a'_j is $\frac{2}{j-i+1}$. □

Using this lemma, the expected number of comparisons is:

$$\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \quad (\text{let } k = j - i) \\
&\le \sum_{i=1}^{n-1} 2H_n \\
&< 2nH_n \approx 2n \ln n = O(n \log n)
\end{aligned}$$

This analysis is very precise, intuitive, and insightful.

2.2 Concentration Bounds: Markov's Inequality

Is knowing the expectation sufficient to call an algorithm "good"? Ideally, we would like to know the full distribution of the running time. Since this is often complicated, we use concentration inequalities (bounds).

Theorem 2 (Markov's Inequality)**.** *Suppose X is a non-negative random variable with finite $E[X]$. Then for any $t > 0$:*

$$Pr[X \geq t \cdot E[X]] \leq \frac{1}{t}$$

For Randomized Quicksort, since $E[X] \approx 2nH_n$, we can say:

$$Pr[\#\text{Comparisons} \geq 10nH_n] \leq \frac{1}{5}$$

Here, $t = 5$, since $10nH_n = 5 \cdot (2nH_n) \approx 5 \cdot E[X]$.

3 Max-Cut

Given a graph $G = (V, E)$, the Max-Cut problem is to find a partition of the vertex set V into two disjoint sets $(S, V - S)$ such that the number of edges crossing the cut (with one endpoint in S and the other in $V - S$) is maximized. This problem is NP-Hard.

3.1 A Simple Randomized Algorithm

1. Initialize $S = \emptyset$.
2. For each vertex $u \in V$, add u to S independently with probability $\frac{1}{2}$.
3. Output the cut $(S, V - S)$.

3.1.1 Analysis

Let X be the number of edges crossing the cut. For each edge $e \in E$, let X_e be the indicator variable that e crosses the cut. Then $X = \sum_{e \in E} X_e$. By linearity of expectation, $E[X] = \sum_{e \in E} E[X_e] = \sum_{e \in E} Pr[X_e = 1]$. For an edge $e = (u, v)$, it crosses the cut if $u \in S$ and $v \notin S$, or if $u \notin S$ and $v \in S$.

$$Pr[X_e = 1] = Pr[u \in S \wedge v \notin S] + Pr[u \notin S \wedge v \in S] = \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$$

Thus, $E[X] = \sum_{e \in E} \frac{1}{2} = \frac{m}{2}$, where $m = |E|$. Since the optimal cut size $OPT \leq m$, this gives a $\frac{1}{2}$-approximation algorithm in expectation: $E[X] \geq \frac{OPT}{2}$.

3.1.2 The Probabilistic Method

This result also shows a graph-theoretic fact: in any graph, there exists a cut of size at least $\frac{m}{2}$. This way of using probability to prove the existence of a combinatorial object is called the **probabilistic method**.

3.2 Reverse Markov Inequality

Is there an analogue of Markov's inequality for lower bounds? A general bound of the form $Pr[X < tE[X]] \leq f(t)$ is not feasible if X has no upper bound. However, if it does, we can prove the following.

Theorem 3 (Reverse Markov Inequality)**.** *Let X be a non-negative random variable such that $X \leq B$ almost surely. Then for any $d < E[X]$:*

$$Pr[X \geq d] \geq \frac{E[X] - d}{B - d}$$

Proof. Apply Markov's inequality to the non-negative random variable $Y = B - X$. □

For Max-Cut, the number of crossing edges $X \leq m$. We have $E[X] = \frac{m}{2}$. Let's find the probability of getting a cut of size at least $(1-\delta)\frac{m}{2}$:

$$Pr[X \geq (1-\delta)\frac{m}{2}] \geq \frac{\frac{m}{2} - (1-\delta)\frac{m}{2}}{m - (1-\delta)\frac{m}{2}} = \frac{\delta \frac{m}{2}}{\frac{m}{2}(1+\delta)} = \frac{\delta}{1+\delta}$$

By repeating the experiment many times and taking the maximum cut found, we can get arbitrarily close to the expected value of $\frac{m}{2}$.

4 Approximating Max-Cut via Vector Programming

The simple randomized algorithm guarantees an expected cut of $\frac{m}{2}$, but it might perform poorly on some instances (e.g., on a bipartite graph where $OPT = m$). For a long time, this was the best approximation ratio known. Goemans and Williamson introduced a new algorithm based on semidefinite programming.

4.1 Problem Formulation and Relaxation

Max-Cut can be formulated as a quadratic program. For each vertex $i \in V = \{1, \ldots, n\}$, let $x_i \in \{-1, 1\}$. We can think of this as assigning each vertex to one side of the cut. An edge (i,j) is in the cut if $x_i \neq x_j$, which means $x_i x_j = -1$. The size of the cut is $\sum_{ij \in E} \frac{1 - x_i x_j}{2}$. So, the problem is:

$$\max \sum_{ij \in E} \frac{1}{2}(1 - x_i x_j) \quad \text{s.t.} \quad x_i \in \{-1, 1\} \quad \forall i \in V$$

This is equivalent to requiring $x_i^2 = 1$. This problem is NP-Hard. The key idea is to relax the constraint that x_i are scalars. Instead, let them be vectors.

$$\text{(SDP Relaxation)} \quad \max \sum_{ij \in E} \frac{1}{2}(1 - \bar{v}_i \cdot \bar{v}_j) \quad \text{s.t.} \quad ||\bar{v}_i|| = 1, \bar{v}_i \in \mathbb{R}^n \quad \forall i \in V$$

This is a semidefinite program (a type of convex optimization problem) and can be solved efficiently to near-optimality. Let OPT_{SDP} be the value of this relaxation. Clearly, $OPT_{SDP} \geq OPT$.

4.2 Goemans-Williamson Rounding

How do we get a cut from the vector solution $\bar{v}_1, \ldots, \bar{v}_n$?

1. Solve the SDP to get vectors $\bar{v}_1, \ldots, \bar{v}_n$.
2. Pick a random hyperplane through the origin. This can be done by picking a random unit vector $\bar{r} \in \mathbb{R}^n$.
3. Partition the vertices into $S = \{i \mid \bar{r} \cdot \bar{v}_i > 0\}$ and $V - S$.

4.2.1 Analysis

Let X be the size of the cut produced by this rounding procedure. $E[X] = \sum_{ij \in E} Pr[\text{edge } ij \text{ is cut}]$. An edge (i,j) is cut if $\bar{v}_i$ and $\bar{v}_j$ fall on opposite sides of the random hyperplane. The probability of this is proportional to the angle between them. Let θ_{ij} be the angle between $\bar{v}_i$ and $\bar{v}_j$.

$$Pr[\text{edge } ij \text{ is cut}] = \frac{\theta_{ij}}{\pi}$$

So, the expected size of the cut is $E[X] = \sum_{ij \in E} \frac{\theta_{ij}}{\pi}$. The value of the SDP solution is $OPT_{SDP} = \sum_{ij \in E} \frac{1}{2}(1 - \cos\theta_{ij})$. It can be shown that for $\theta \in [0, \pi]$:

$$\frac{\theta}{\pi} \geq 0.878 \cdot \frac{1}{2}(1 - \cos\theta)$$

This implies $E[X] \geq 0.878 \cdot OPT_{SDP} \geq 0.878 \cdot OPT$.

4.3 Generating Random Vectors

How do we generate a random unit vector $\bar{r}$ in a high-dimensional space $\mathbb{R}^d$?

1. Generate a vector $\bar{z} = (X_1, X_2, \ldots, X_d)$, where each X_i is an independent random variable drawn from the standard normal distribution $N(0, 1)$.
2. The joint probability density function is $f_{\bar{z}}(x_1, \ldots, x_d) = \frac{1}{(\sqrt{2\pi})^d} e^{-(\sum x_i^2)/2}$.
3. This distribution is centrally symmetric, meaning the density only depends on the length of the vector $||\bar{x}||$. Therefore, the direction of the vector $\bar{z}$ is uniformly distributed.
4. Output the unit vector $\bar{r} = \frac{\bar{z}}{||\bar{z}||}$.