Guide to Randomized Algorithms

Randomized algorithms are often easier to design than deterministic algorithms, though often the analysis requires some manipulations of random events or random variables.

This handout contains a few sample randomized algorithms and solutions so that you can get a better sense for how to approach solving problems with randomness.

Sample Problem: Housing Horrors

Consider the following problem: you have k housing complexes and n total people to place into them. The problem is that some pairs of people really, *really* don't like one another and have made it clear that they don't want to be put into the same housing complex. Given a list of m constraints, design an algorithm to assign people to houses in a way that minimizes the number of constraints violated.

It turns out that this problem is **NP**-hard, so it's probably going to be difficult (if not impossible) to get an exact answer. Fortunately, we can design a randomized algorithm that, on expectation, will get a large fraction of the constraints satisfied.

When facing an **NP**-hard optimization problem, it's often useful, as an initial approach, to guess a totally random answer and see how well it does. Let's see what happens if we do that.

Algorithm: Assign people to houses uniformly at random.

To analyze this algorithm, we can set up a random variable, which we'll call X, that represents the total number of constraints that we can satisfy. If we're interested in the average number of constraints that we can satisfy, we want to know $E[X]$. For notational simplicity, let's denote by S the set of all constraints, and say $(i, j) \in S$ iff person i and person j should be placed separately.

As with many of the other randomized analyses we've seen so far, we'll try to write the random variable X as the sum of other random variables. That way, we can express $E[X]$ as the sum of the expected values of other random variables using linearity of expectation. In our case, we can create an indicator random variable C_{ij} for each constraint saying that person i and person j should be kept separate, where $C_{ij} = 1$ if they are placed into different locations and $C_{ij} = 0$ otherwise. Then, we have that

$$X = \sum_{(i,j)\in S} C_{ij}$$

Therefore, by linearity of expectation:

$$\begin{aligned} E[X] &= E[\sum_{(i,j)\in S} C_{ij}] \\ &= \sum_{(i,j)\in S} E[C_{ij}] \\ &= \sum_{(i,j)\in S} P(i \text{ and } j \text{ are placed separately}) \end{aligned}$$

So now if we can determine the probability that person i and person j are placed in separate housing, we can determine E[X]. Since we're assigning people completely randomly, the probability that i and j are placed into the same house is $1 / k$. Therefore, the probability that i and j are not placed into the same house is $(k - 1) / k$. Therefore:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{(i,j)\in S} P(i \text{ and } j \text{ are placed separetely}) \\ &= \sum_{(i,j)\in S} \frac{k-1}{k} \\ &= \frac{m(k-1)}{k} \end{aligned}$$

In other words, on expectation, we can respect a $(k - 1) / k$ fraction of the total number of constraints. The maximum possible number of constraints we can satisfy is m, so our algorithm will always produce an answer that is within a $(k - 1) / k$ fraction of optimal. The above line of reasoning is pretty much what we could write as a correctness proof (asserting we were within a factor of $(k - 1) / k$ of optimal on expectation), assuming that we clean up some of the informal language.

Here is another sample problem; the solution is on the next page.

The Majority Element Problem Revisited

Suppose you are interested in solving the majority element problem from the previous problem set using a randomized algorithm. Recall that in this problem, you want to try to determine whether there are strictly more than $n / 2$ elements in an array with the same value, subject to the constraint that you can only learn whether two elements are equal or different.

Design an O(n)-time randomized algorithm where if there is a majority element, your algorithm returns one with probability at least $1 - 10^{-9}$, and if there is no majority element, your algorithm always returns "no majority."

Solution to The Majority Element Problem Revisited

The key insight behind the solution to this problem is that we can start with an algorithm that has a modest chance of success, then iterate it enough times to drive the error rate down to less than one in a billion. This particular algorithm works by starting with just over a 50% chance of success, then iterating it 30 times to amplify the probability to at least $1 - 10^9$ (since $(1 / 2)^{30} \approx 10^{-9}$).

Algorithm: Repeat this process 30 times: choose an element uniformly at random, then compare that element to every other element in the array. If more than $n / 2 - 1$ elements compare equal to the element, return that element. If after 30 iterations this process has not returned a majority, return "no majority."

Correctness:

Theorem: If there is no majority element, our algorithm always returns "no majority."

Proof: The algorithm only returns an element x if it finds that more than $n / 2 - 1$ elements in the array are equal to x, meaning that more than $n / 2$ total elements are equal to the element x. Consequently, x is a majority element. Therefore, if there is no majority element, no element chosen will be a majority, so after 30 iterations the algorithm will return "no majority." ■

Theorem: If there is a majority element, it will be returned with probability at least $1 - 10^{-9}$.

Proof: Let $\mathcal{E}$ be the event that our algorithm does not return a majority when one exists. Let $\mathcal{E}_1$, $\mathcal{E}_2$, …, $\mathcal{E}_{30}$ be the events that on iterations 1, 2, 3, …, 30, the algorithm does not find a majority element. This means that

$$\mathcal{E} = \bigcap_{i=1}^{30} \mathcal{E}_i$$

Using the chain rule for conditional probability:

$$P(\mathcal{E}) = P(\mathcal{E}_1) \cdot P(\mathcal{E}_2 | \mathcal{E}_1) \cdot P(\mathcal{E}_3 | \mathcal{E}_2, \mathcal{E}_1) \cdot \ldots P(\mathcal{E}_{30} | \mathcal{E}_{29}, \mathcal{E}_{28}, \ldots, \mathcal{E}_1)$$

Assuming iterations $k - 1$, $k - 2$, …, 1 of the algorithm all failed to find a majority element, the probability that iteration k fails to return a majority element is the probability that on iteration k the algorithm chooses a non-majority element. Since a majority element exists and we choose elements uniformly at random, this occurs with probability less than 1/2. Consequently, we have

$$P(\mathcal{E}) < \prod_{i=1}^{30} \frac{1}{2} = \left(\frac{1}{2}\right)^{30}$$

Since $(1 / 2)^{30} < 10^{-9}$, this means that the probability that the algorithm does not find a majority if one exists is at least $1 - (1 / 2)^{30} > 1 - 10^{-9}$. ■

Runtime:

The algorithm runs for 30 iterations and on each iteration does $\Theta(n)$ work picking a random element and comparing each element against it. Consequently, the total runtime is $O(n)$.