CS 466/666 Algorithm Design and Analysis, Spring 2019. Waterloo.

Lecture 1: Introduction

First, we study the string equality to see the power of randomness in computation.

Then, we see an overview of the course and also the course information.

Finally, we review linearity of expectation and apply it to analyze randomized quicksort.

Course homepage: https://cs.uwaterloo.ca/~lapchi/cs466

String equality [MR 7.4] This is a striking use of randomness in computation.

Suppose a company maintains multiple copies of the same huge data set.

From time to time they want to check that the copies are still the same.

Think of a data set is an n-bit string, for some very large n.

Problem: Alice has a binary string $a_1 a_2 \ldots a_n$. Bob has a binary string $b_1 b_2 \ldots b_n$.

They would like to check whether their strings are equal, with as few communications as possible.

Deterministic algorithms: An obvious algorithm is for Alice to send all her bits to Bob, then Bob can check and tells Alice the answer. But it requires $n+1$ bits of communications.

As you might imagine, there are no better ways to do it, and it can be proved that there are no deterministic algorithms that can do better, using techniques in communication complexity.

Randomized algorithms: Surprisingly, there is a randomized algorithm for this problem using only $O(\log n)$ bits.

There are different ways to solve the problem.

One way is to consider the polynomials $A(x) = \sum_{i=1}^{n} a_i x^i$ and $B(x) = \sum_{i=1}^{n} b_i x^i$.

Algorithm:
- Alice and Bob agrees on a large enough field F (e.g. modular arithmetic over a large enough prime p).
- Alice picks a random element $r \in F$, evaluate $A(r)$ and send r and $A(r)$ to Bob.
- Bob computes $B(r)$, and return "consistent" if $A(r) = B(r)$, and "inconsistent" if $A(r) \neq B(r)$.

That's the algorithm. We need to analyze its success probability.

Analysis: If the two strings are the same, then $A(x) \equiv B(x)$ and the algorithm will always says "consistent".

So, when the algorithm says "inconsistent", it will never make a mistake.

But the algorithm may make mistake when the two strings are not the same and it says "consistent".

We would like to upper bound this error probability.

If the two strings are not the same, then $A(x) \not\equiv B(x)$ but we have chosen r s.t. $A(r) = B(r)$.

What is this error probability? It only happens when r is a root of $(A-B)(x)$.

Since $A(x)$ and $B(x)$ are polynomials of degree at most n, so is $(A-B)(x)$.

There are at most n roots of a degree n polynomial.

Since we pick a random r, the error probability is at most $n/|F|$.

If we choose $|F| \approx 1000n$, then the error probability is at most 0.001.

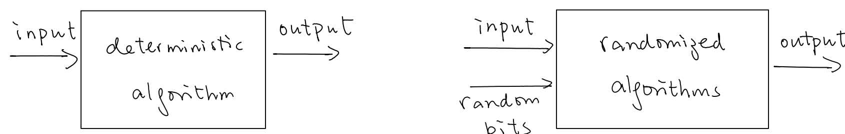
And the algorithm only needs to send $O(\log |F|) = O(\log n)$ bits.

If we would like to decrease the error probability further, one standard way is to repeat the algorithm a few times.

In this case, however, a better way is simply to pick a large field size, e.g. if $|F| = n^2$, then the error probability is at most $\frac{1}{n}$, while we still only send $O(\log n)$ bits.

Randomized Algorithms

Informally, randomized algorithms are algorithms that flip coins.



In deterministic algorithms, given an input, it returns the same output all the time.

In randomized algorithms, the output is not only a function of the input, but also a function of the random bits. So the output is a random variable. Also the running time is a random variable.

Why would flipping coins help?

The main difference is that we allow the algorithm to make errors sometimes, and only requires that the output is correct with high probability. Or, we allow the randomized algorithm to be slow sometimes, but it is fast with high probability. Think of high probability as 99.99999999%.

Another way to think of a randomized algorithm is to see it as a "family of deterministic algorithms", one for each random string. Then, the requirement that the randomized algorithm should succeed

with 99% is equivalent to requiring that for each input 99% in the family should succeed.

So, instead of requiring a single deterministic algorithm to be correct for all inputs, we allow ourselves to have a family of algorithms such that for each input most of them are correct.

Perhaps surprisingly, this tradeoff could bring us a significant improvement in the running time, and sometimes could achieve something impossible by deterministic algorithms.

The following are some different ways that randomness can help.

Faster and simpler algorithms: Sometimes deterministic decisions are difficult or expensive to make (e.g. finding a non-ro

Random decisions or random sampling can often achieve a similar guarantee (with small error probability), and these can speedup and/or simplify the algorithms.

We will see several interesting problems for which the best known randomized algorithms are significantly faster and simpler than the best known deterministic algorithms.

Some examples include graph algorithms for MST and minimum-cut.

A notable example is approximate counting, for which we have randomized polytime algorithms but we do not know of deterministic polynomial time algorithms yet.

In many cases, the randomized algorithms are also much simpler, much easier to be implemented.

Avoiding communications: In distributed and parallel computing, often much communications are needed to guarantee that we always have the correct answers. On the other hand, if we make random decisions locally, in many cases we can prove that bad events happen with low probability.

We will see how to compute matchings in parallel and how to do network coding in a distributed manner, where we do not know how to achieve the same using deterministic algorithms.

Protecting from adversary: When we analyze deterministic algorithms, we assume that there is an adversary who knows exactly what we do and then provide the worst input for our algorithm.

These make some tasks impossible to do deterministically, e.g. online algorithms, sublinear algorithms.

In the analysis of randomized algorithms, we assume that the adversary does not know our random bits in advance, and we can prove much stronger results with this assumption.

Some examples that we will see include hashing, data streaming, and sublinear time algorithms.

(The string equality problem is also one such example.)

Probabilistic methods: Interestingly, probabilistic methods can be applied to prove statements that have nothing to do with probabilities at all, which are very difficult to be proved otherwise. Some examples include the existence of good graphs (expander graphs), good codes (rate achieving error-correcting codes), good solutions (local lemma).

We will see some examples of each category above in this course.

Let me emphasize that randomized algorithms could do much better because of weaker requirements (e.g. to allow errors) or stronger assumptions (e.g. private randomness).

But in almost all scenarios this tradeoff is preferrable and the assumption is reasonable.

Course information

The course outline is posted on the course homepage. Detailed information can be found there.

The following is a list of background knowledge that is required for this course.

- first course in discrete math (induction, contradiction, asymtotics, graphs).
- first course in algorithms (time complexity, recursion, reductions, basic data structures and algorithms).
- first course in probability (random variables, Gaussians, expectation, variance).

I encourage you to solve the problems in homework 0 (no need to submit) to test your basic probability skills.

Requirements: homework 40% (5 assignments)

midterm 20% (June 19, 7-8:50pm)

final exam 40%

Request: Try not to use electronic devices in lectures, especially playing games / browsing / facebook / email are not allowed. Reading or taking notes are okay.

Quick basic probability review [MU, chapter 1 and 2]

Sample space Ω: set of all possible outcomes, each with a "probability" associated with it.

Event E: subset of outcomes. $\Pr(E)$ = sum of the probabilities of its outcomes

Axioms: ① $0 \le \Pr(E) \le 1$ ② $\Pr(\Omega) = 1$ ③ $\Pr(\bigcup_i E_i) = \sum_i \Pr(E_i)$ for disjoint E_i.

Union bound: $\Pr(\bigcup_i E_i) \le \sum_i \Pr(E_i)$

Inclusion-exclusion principle: $\Pr(\bigcup_i E_i) = \sum_i \Pr(E_i) - \sum_{i,j} \Pr(E_i \cap E_j) + \sum_{i,j,k} \Pr(E_i \cap E_j \cap E_k) - \dots$

$$+ (-1)^{\ell+1} \sum_{i_1 < i_2 < \dots < i_\ell} \Pr\left(\bigcap_{r=1}^{\ell} E_{i_r}\right) \dots$$

Conditional probability: $\Pr(E|F) = \Pr(E \cap F) / \Pr(F)$

Independence: $\Pr(\cap E_i) = \prod_i \Pr(E_i)$

Total probability: Let E_i be disjoint and $\bigcup_i E_i = \Omega$.

Then $\Pr(B) = \sum_i \Pr(B \cap E_i) = \sum_i \Pr(B|E_i)\Pr(E_i)$.

Baye's law: Let E_i be disjoint and $\bigcup_i E_i = \Omega$.

Then $\Pr(E_j|B) = \dfrac{\Pr(B|E_j)\Pr(E_j)}{\sum_i \Pr(B|E_i)\Pr(E_i)}$

Random variable X is a function from $\Omega \to \mathbb{R}$ $\Pr(X=a) = \sum_{s \in \Omega : X(s) = a} \Pr(s)$.

Independence X and Y are independent if and only if $\Pr(X=x \cap Y=y) = \Pr(X=x) \cdot \Pr(Y=y)$.

Expectation $E[X] = \sum_i i \Pr(X=i)$.

Linearity of expectation $E[\sum_i X_i] = \sum_i E[X_i]$.

It is important to note that this holds even for dependent variables, e.g. $E[X_i] + E[X_i^2] = E[X_i + X_i^2]$.

Conditional expectation $E[Y|Z=z] = \sum_y y \Pr(Y=y | Z=z)$

$E[Y|Z]$ is a random variable of Z that takes on the value $E[Y|Z=z]$ if $Z=z$.

Please review some basic random variables such as binomial and geometric random variables [MU 2.2-2.4].

Randomized Quicksort [MU 2.5]

Task: Given n distinct numbers $x_1, x_2, \dots, x_n$, output the numbers in sorted order.

Algorithm: ① Pick a random number x from $\{x_1, \dots, x_n\}$.

② Construct the sets $S_{<x}$ and $S_{>x}$.

③ Return $\{\text{quicksort}(S_{<x}), x, \text{quicksort}(S_{>x})\}$.

Intuition: Most of the time, we break the sequence "evenly" (not necessarily $(n/2, n/2)$, enough to have $(\frac{n}{10}, \frac{9n}{10})$ most of the time).

Then the recursion depth is $O(\log n)$, and the total running time is $O(n \log n)$.

It can be analyzed rigorously using a probabilistic recurrence (e.g. see [MR 1.4]).

Linearity of expectation There is a simple and elegant analysis using linearity of expectation.

The key point is to define the random variables cleverly.

Let $y_1, y_2, \ldots, y_n$ be the sorted sequence.

Let X_{ij} be the indicator variable such that $X_{ij} = \begin{cases} +1 & \text{if } y_i \text{ and } y_j \text{ are compared.} \\ 0 & \text{otherwise.} \end{cases}$

Let X be the total number of comparisons, i.e. the total running time of the algorithm.

Then $E[X] = E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}]$ by linearity of expectation

$= \sum_{i<j} \Pr[y_i \text{ and } y_j \text{ are compared}]$

Note that y_i and y_j are compared iff y_i or y_j is selected as the random number (pivot), and $y_{i+1}, \ldots, y_{j-1}$ are not selected yet.

Since each number is selected with equal probability, this happens with $2/(j-i+1)$.

So, $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(j-i+1) \leq 2n \log n$.

Monte Carlo, Las Vegas, RP, BPP [MR 1.2]

Monte Carlo: may make mistakes

It is important to note that the error is independent on the input, but just dependent on the random bits, i.e. works on every input, e.g. string equality.

Las Vegas: no mistakes, but running time is a random variable, e.g. randomized quicksort.

Note that a Monte Carlo algorithm can be changed to a Las Vegas algorithm if there is a quick algorithm to check whether the answer is correct. It is often not easy to have such a quick checking algorithm. A known example is a linear time minimum spanning tree verification algorithm (deterministic, very nontrivial), and this can turn a Monte Carlo linear time MST algorithm into Las Vegas.

RP (randomized polynomial time): one sided error

YES-instance $\Rightarrow \Pr(\text{algorithm says YES}) \geq \frac{1}{2}$

NO-instance $\Rightarrow \Pr(\text{algorithm says YES}) = 0$

BPP (Bounded-error probabilistic polynomial time) : two sided error

YES - instance $\Rightarrow$ $\Pr$(algorithm says YES) $\geq \frac{1}{2} + \varepsilon$

NO - instance $\Rightarrow$ $\Pr$(algorithm says YES) $\leq \frac{1}{2} - \varepsilon$.

Can take majority to boost the success probability.

<u>Conjecture</u> : P = BPP . That is, every randomized algorithm can be derandomized.

<u>Homework</u> : Do homework 0.

Review basic probability knowledge (e.g. read chapter 1 and 2 of [MU]).