Randomized Algorithms

We already learned quite a few randomized algorithms in the online algorithm lectures. For example, the MARKING algorithm for paging was a randomized algorithm; as well as the Randomized Weighted Majority. In designing online algorithms, randomization provides much power against an oblivious adversary. Thus, the worst case analysis (the worst case of the expectation) may become better by using randomization. We also noticed that randomization does not provide as much power against the adaptive adversaries.

We'll study some examples and concepts in randomized algorithms. Much of this section is based on (Motwani and Raghavan, Randomized Algorithm, Chapters 1, 5, 6).

Random QuickSort

Suppose A is an array of n numbers. Sorting puts the numbers in ascending order. The quick sort is one of the fastest sorting algorithm.

QuickSort:

1. If A is short, sort using insertion sort algorithm.
2. Select a "pivot" $x \in A$ arbitrarily.
3. Put all members $\leq x$ in A_1 and others in A_2.
4. QuickSort A_1 and A_2, respectively.

Suppose the choice of x is such that $|A_1| = |A_2| = \frac{|A|}{2}$, then the time complexity is $T(n) = n + 2 \cdot T\left(\frac{n}{2}\right)$. By using Master's theorem, $T(n) = O(n \log n)$. This is even true for $\frac{|A|}{4} \leq |A_1| \leq \frac{3|A|}{4}$.

For worst case analysis, QuickSort may perform badly because x may be such that A_1 and A_2 are awfully imbalanced. So the time complexity becomes $O(n^2)$. If we don't like average analysis, which is venerable to a malicious adversary, we can do randomization.

RandomQS: In QuickSort, select the pivot x uniformly at random from A.

Theorem 1: The expected number of comparisons in an execution of RandomQS is at most $2nH_n$, where H_n is the n-th harmonic number.

Proof: Clearly any pair of elements are compared at most once during any execution of the algorithm. Let $A(i)$ be the i-th ranked element in A. Let $X_{i,j} = 0,1$ be an indicator that whether elements $A(i)$ and $A(j)$ are ever compared directly against each other during the algorithm. Let $p_{i,j} = \Pr(X_{i,j} = 1)$.

Notice in any step of the RandomQS algorithm, if the pivot x is not one of $A(i), A(i+1), \dots, A(j)$, then the iteration is irrelevant to the event $X_{i,j}$. The only thing that is relevant is that one of $A(i), A(i+1), \dots, A(j)$ is selected as a pivot. If $A(i)$ or $A(j)$ is selected, then they are

compared directly. If any other element in this series is selected, then $A(i)$ or $A(j)$ will never be compared directly.

As long as none of $A(i), A(i+1), \ldots, A(j)$ is selected as a pivot yet, they are to be selected at equal probability. Thus, the probability that one of $A(i)$ and $A(j)$ is selected first is $\frac{2}{j-i+1}$. That is to say $p_{i,j} = \frac{2}{j-i+1}$. Therefore,

$$E[\#comparisons] = E\left[\sum_{i=1}^{n}\sum_{j>i} X_{i,j}\right] = \sum_{i=1}^{n}\sum_{j>i} p_{i,j} = \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1} \leq \sum_{i=1}^{n}\sum_{k=1}^{n}\frac{2}{k} \leq 2nH_n$$

QED.

Note that $H_n \approx \log n$.

Comments on Randomization

Randomized algorithms vs. Deterministic algorithms

For each fixed input, the algorithm may perform differently from run to run. The variation comes from the deliberate random choices made during the execution of the algorithm; but not from a distribution assumption of the input. Thus the expected performance holds for *every* input. This is different from average analysis, which averages costs over a set of inputs.

The performance of a randomized algorithm does not rely on the intention of the adversary (such as a hacker), but instead relies on pure chance.

We've seen examples where randomization really made a difference for online algorithm against an oblivious adversary. But for offline algorithms, in general people do not know if randomization indeed can beat the best possible (but not yet known) deterministic algorithm. After all, a polynomial time randomized algorithm with input I is just a deterministic algorithm with input I and a series of random bits R with length bounded by a polynomial. Does knowing a sequence of random bits really help?

Nevertheless, in many cases randomization helps to design

1. simpler algorithms, and
2. more efficient algorithms *than the best known deterministic* algorithm.

We will see more of such examples in this chapter.

Random Bits

Although it is easy to say an algorithm can make random choices, it is not as easy to obtain a true random bit in a computer. Traditionally random numbers are generated with physical devices such as a dice in gambling. But these are too slow and too inconvenient for computation.

Thus, a computer often uses a pseudo random number generator to produce an integer sequence seemingly random. But it's not truly random. For example, the Java Random class uses an integer random seed. Each seed produces a unique deterministic sequence of pseudo-random numbers. This is clearly not random because it can only give at most 2^{64} possible sequences. A good example of this is that many people do not reset the default password a system automatically generated for them. A careless implementation of the key generation with a pseudo random number generator will make it easy to break the key.

Another source of random is to convert noises in physical world to digital signals. For example, RANDOM.org uses atmospheric noises to provide random number generation service to the public. It has generated 1.82 trillion random bits for the Internet community as of August 2014. Another example is some hardware random number generators (TRNG, or true random number generator) that are commercially available. These TRNG can utilize thermal noise, the photoelectric effect, or other quantum phenomena to produce random numbers.

Even if we can generate random bits, there are problems. For examples:

1. What if the physical device generates 0 and 1 with skewed probability? ($p_0 \neq p_1$)
2. What if we want a uniform distribution from 1 to not-a-power-of-2?

But for most non-critical applications, a pseudo random number generator is good enough.

Random Median Selection

Recall that we had a linear time algorithm to select the k-th largest element from a list. The algorithm is

Algorithm Select(L, k): L is the list and k is the rank.

1. If L is short then use sorting algorithm and return.
2. Find a "pivot" number x. Put all elements $\geq x$ in list A, and all elements $< x$ in list B.
3. If $|A| > k$ then return Select(A, k).
4. Else return Select(B, k-|A|).

The trickiest part was the finding of the pivot x. In order to make the two lists A and B approximately balanced, we had to divide the list into $\frac{n}{5}$ groups of 5-elements each, and then do a median of medians.

With randomization, we might guess that if we randomly select a pivot from the list, chances are that p divide the list fairly evenly. So we have a randomized algorithm for Select-K.

RandomSelect(L,k): L is the list and k is the desired rank.

1. If L is short then use sorting algorithm and return.
2. Randomly select a "pivot" number x. Put all elements $\geq x$ in list A, and all elements $< x$ in list B.

3. If $|A| > k$ then return RandomSelect(A, k).
4. Else return RandomSelect(B, k-|A|).

Theorem: Algorithm RandomSelect has an expected time complexity of $O(n)$.

Proof: Suppose the expected time complexity is $E[T(n)]$.

$$E[T(n)] \leq \frac{1}{n} \cdot \left(\sum_{i=1}^{\frac{n}{2}} E[T(n-i)] + \sum_{i=\frac{n}{2}+1}^{n} E[T(i)] \right) + \alpha n \leq \frac{2}{n} \cdot \sum_{i=1}^{\frac{n}{2}} E[T(n-i)] + \alpha n$$

The last αn is for the $O(n)$ overhead. We prove the linearity by induction. The base case $n = 1$ is trivial. Suppose $E[T(k)] \leq c \cdot k$ holds for a constant c and $k < n$. Then

$$E[T(n)] \leq \frac{2}{n} \cdot \sum_{i=1}^{\frac{n}{2}} E[T(n-i)] + \alpha n \leq \frac{2c}{n} \cdot \sum_{i=1}^{\frac{n}{2}} (n-i) + \alpha n = \frac{2c}{n} \cdot \frac{\left(n + \frac{n}{2} - 1\right) n}{4} + \alpha n$$
$$\leq \frac{3c}{4} n + \alpha n$$

So, as long as we let $c \geq 4\alpha$, we have $E[T(n)] \leq c \cdot n$.

QED.

Random Min-Cut

Given a graph $G = \langle V, E \rangle$, a cut is a partition of the vertex set V into two disjoint sets V_1 and V_2, such that $V_1 \cup V_2 = V$. An edge is cut if its two vertices fall in V_1 and V_2, respectively. The Min-Cut problem asks for a cut so that the minimum number of edges are cut.

In a connected graph (such as computer network, or operation network of an organization), a cut breaks the connectivity. A min cut minimize the cost to achieve this goal.

Deterministic Algorithm

A popular deterministic algorithm is to reduce the Min-Cut problem to Maximum Flow. The so called s-t Cut problem is to find a Min-Cut so that the two given vertices s and t fall in the two vertex sets V_1 and V_2, respectively.

Theorem 2: The s-t Min-Cut size is equal to the maximum flow from s to t.

Proof omitted.

To find min-Cut, one can fix an arbitrary vertex s and try every other t. Then choose the minimum. Thus, this algorithm needs at most $O(|V| \cdot T)$ time, where T is the time complexity of the maximum flow algorithm.

There are also algorithms that do not rely on Maximum-Flow. One of such algorithms runs in $O(|V| \times |E| + |V|^2 \times \log|V|)$. (See. Stoer and Wagner, Journal of the ACM, Vol. 44, No. 4, July 1997, pp. 585–591.)

Randomized Algorithm I

This simple algorithm was first published by D. R. Karger in 1994 when he was a Ph.D. student at Stanford. (Random sampling in cut, flow, and network design problems. Proc. 25th STOC, 648–657, 1994.)

A *multigraph* is just a graph that allows multiple edges connecting a pair of vertices.

Karger's algorithm is based on a simple edge contraction operation. When an edge (u, v) is contracted, they are replaced with a single new node w. Edges incident to either u or v are now incident with the new node w. Note that contraction can create edges from w to w. Such self-loops are removed from the graph. Also the contraction can create multiple edges between two vertices. These are all kept and regarded as different edges. Thus, the algorithm works on a multigraph.

Karger's Algorithm: Keep choosing an edge uniformly at random and contracting it; until there are only two vertices left. Output the two sets of vertices that were contracted to these two remaining vertices.

Note that for any Min-Cut V_1, V_2, the algorithm gives the same cut as the output with a positive probability: if and only if none of the contracted edges belong to the cut edge set.

Theorem 3: The probability that Karger's algorithm outputs the Min-Cut is at least $\frac{2}{n\cdot(n-1)}$, where $n = |V|$.

Proof: When there are n vertices in the multi-graph, if the Min-Cut has size k, then each vertex has degree at least k. So, the number of edges is at least $\frac{n\cdot k}{2}$. Therefore, the probability that the selected edge is not in the Min-Cut is $\frac{n-2}{n}$. Thus, the probability that none of the contraction is in the Min-Cut is at least

$$\prod_{i=3}^{n} \frac{i-2}{i} = \frac{2}{n(n-1)}$$

QED.

The running time is $O(n^2)$ because there are $O(n)$ contractions and each contraction takes at most $O(n)$ time.

In order to achieve a high probability, we only need to repeat the algorithm $100n(n-1)$ times. Notice that the probability that none of the repetition worked is at most

$$\left(1-\frac{2}{n\cdot(n-1)}\right)^{100n(n-1)}=\left(\left(1-\frac{2}{n\cdot(n-1)}\right)^{\frac{n(n-1)}{2}}\right)^{200}\approx\left(\frac{1}{e}\right)^{200}$$

The total running time would be $O(n^4)$ in order to achieve high probability because it needs to be repeated $O(n^2)$ times.

Rnadomized Algorithm II

Karger's algorithm was very simple but the running time $\mathrm{O}(\mathrm{n}^4)$ is not as good as the best known deterministic algorithm. This section we improve it so that it runs in $\mathrm{O}(\mathrm{n}^2)$ time. The idea belongs to (D. R. Karger and C. Stein. Proc. 25th STOC, 757–765, 1993).

Re-examine the contraction process. The contraction when there are i vertices is safe (selects an edge outside of the Min-Cut) with probability $\frac{i-2}{i}$. This probability is high at the beginning of the contraction, but reduces significantly towards the end of the contraction. But when we repeat the randomized algorithm, the first half and second half of the contractions are treated the same. A better way is to repeat the second half more times.

The probability that the first $n-\left\lceil\frac{n}{\sqrt{2}}\right\rceil-1$ contractions are all safe is

$$\prod_{i=\left\lceil\frac{n}{\sqrt{2}}\right\rceil+1}^{n}\frac{i-2}{i}=\frac{\left(\left\lceil\frac{n}{\sqrt{2}}\right\rceil+1\right)\left(\left\lceil\frac{n}{\sqrt{2}}\right\rceil\right)}{n(n-1)}\geq\frac{1}{2}$$

Now we give the algorithm. Let G be a multi-graph.

BetterCut(G):

1. Contract multi-graph G to graph G' that has $\left\lceil\frac{n}{\sqrt{2}}\right\rceil+1$ vertices.
2. $X_1\leftarrow$ BetterCut(G')
3. $X_2\leftarrow$ BetterCut(G')
4. Return the better of X_1 and X_2 .

It seems that we are doing the same thing twice in steps 2 and 3. But since it is randomized algorithm, X_1 and X_2 may be different result.

Theorem 4: BetterCut computes the Min-Cut with probability at least $\frac{1}{2\log_2 n}$.

Proof: Let $P(n)$ denote the probability that BetterCut(G) finds the MIN-Cut. This is the case if and only if: (a) step 1 must not contract any edge in the MIN-Cut; and (b) one of the two steps 2 and 3 finds the MIN-Cut. Therefore,

$$P(n)\geq\frac{1}{2}\times\left(1-\left(1-P\left(\frac{n}{\sqrt{2}}\right)\right)^2\right)=P\left(\frac{n}{\sqrt{2}}\right)-\frac{1}{2}\times P\left(\frac{n}{\sqrt{2}}\right)^2$$

Note that we ignored the ceiling operation, and removed the +1 from the size of the smaller graph $\left\lceil \frac{n}{\sqrt{2}} \right\rceil + 1$ – just to make the proof easier to read. Even with them, things are okay.

With this recurrence relation, we can prove by induction that $P(n) \geq \frac{1}{2\log_2 n}$. The base is $P(2) = 1 > \frac{1}{2\log_2 2}$. Notice that by replacing $P\left(\frac{n}{\sqrt{2}}\right)$ with a smaller value $\frac{1}{2\log_2\left(\frac{n}{\sqrt{2}}\right)}$, the right hand side of the inequality (the expression in the middle) is reduced. Thus,

$$P(n) \geq \frac{1}{2\log_2\left(\frac{n}{\sqrt{2}}\right)} - \frac{1}{2} \times \left(\frac{1}{2\log_2\left(\frac{n}{\sqrt{2}}\right)}\right)^2$$

One can verify that when $n \geq 2$, the right hand side is no less than $\frac{1}{2\log_2 n}$.

QED.

Now the success probability is much larger than Karger's algorithm. In order to achieve close to 1 probability, we only need to repeat the algorithm $100 \cdot \log_2 n$ times.

Time complexity for one run of the algorithm is:

$$T(n) = n^2 + 2 \cdot T(\frac{n}{\sqrt{2}})$$

By Master's theorem, $a = 2, b = \sqrt{2}, c = 2$, thus $T(n) = O(n^2 \log n)$.

So, the time needed by the algorithm to achieve positive constant probability is $O(n^2 \log^2 n)$. This is better than any currently known deterministic algorithm to compute Min-Cut.

Las Vegas and Monte Carlo

Randomized algorithm takes chances. In the randomized quick sort algorithm, the algorithm guarantees to have a correct result, but takes chances on the running time. The running time guarantee is given on the "expected time". This type of randomized algorithm is called *Las Vegas algorithm*. In contrast, the randomized Min-Cut algorithm guarantees to finish in a certain amount of time, but takes chances on the correctness of the result. The correctness guarantee is given with "high probability". This type of randomized algorithm is called *Monte Carlo algorithm*. This relates to the facts that there are casinos in these two places. The name 'Monte Carlo' was first used in a paper published by Metropolis and Ulam in 1949. Later on, Las Vegas was used in parallel to describe the different behaviors of the two types of randomized algorithms.

There are actually three types:

- Las Vegas: No error. Running time varies.

- Monte Carlo with one-sided error: Zero error for at least one of the possible outputs (yes/no) that it produces.
- Monte Carlo with two-sided error: Errors for both possible outputs.

So, which one is better? It depends on applications: Critical software (e.g. OS) v.s. real time.

Theorem 5: Every Las Vegas algorithm can be converted to a Monte Carlo algorithm.

Proof is given as an assignment.

Matrix Multiplication Test

Recall that the matrix multiplication $C = A \times B$ takes $O(n^3)$ time for a straightforward algorithm, and takes $O(n^{2.81})$ for a divide and conquer. The asymptotically fastest algorithm currently known takes $O(n^{2.23728639})$. But the algorithms beyond the $O(n^{2.81})$ starts to get complicated. How do you ensure that the program computes the right answer? Software testing during development is all right but we can't cover all the instances. Can we test after the software is deployed, and for each $A \times B$ it computes?

Note that one cannot rely on a slower but simpler algorithm to test in this case, because that makes the initial effort to code the faster algorithm meaningless. The testing will have to run as fast as or faster than computation of $C = A \times B$. There is a very interesting randomized test for this.

Let $x = (x_1, x_2, \dots, x_n)^T$ be a column vector. By basic linear algebra, if $C = A \times B$, then

$$C \cdot x = (A \cdot B) \cdot x = A \cdot (B \cdot x)$$

Note that this can be checked in $O(n^2)$ time for any given x. So, if the computation is correct, then such a check will always return "yes". The question is when the computation is wrong, can we get a "no" answer, with a positive probability?

Lemma: Suppose M is a $n \times n$ matrix that is nonzero, and $x = (x_1, x_2, \dots, x_n)^T$ is a random column vector sampled by assigning each x_i to be 0 or 1, with uniform probability, independently. Let $y = Mx$. Then

$$\Pr(y \neq 0) \geq \frac{1}{2}$$

Proof: Suppose $M_{i,k} \neq 0$. Consider the i-th row of vector y:

$$y_i = \sum_{j=1}^{m} M_{i,j} \cdot x_j = \left(\sum_{j \neq k} M_{i,j} \cdot x_j \right) + M_{i,k} \cdot x_k$$

Regardless of the value of $\sum_{j \neq k} M_{i,j} \cdot x_j$, since x_k takes 0 and 1 independently to other x_j, the above value is nonzero with at least probability $\frac{1}{2}$.

QED

With the lemma, we obtain a randomized algorithm to test whether $C = A \times B$.

MultiplicationTest(A,B,C)
1. Select a random 0-1 vector x.
2. Return $C \cdot x == A \cdot (B \cdot x)$

When the true answer is yes, the algorithm always return yes. When the true answer is no, the algorithm returns no with probability at least $\frac{1}{2}$. So this is a Monte-Carlo algorithm with one-sided error. We can repeat the test multiple times to increase the probability.

Max-Cut and Probabilistic Method

The Max-Cut problem is similar to Min-Cut. But Mxax-Cut's goal is to maximize the number of edges that are cut. The Max-Cut problem is an NP-hard problem. So, there is unlikely a polynomial time algorithm for it. We study an interesting theorem for it.

Max-Cut Lemma: In any graph $G = \langle V, E \rangle$, the max-cut size is at least $\frac{|E|}{2}$.

Proof: For each vertex, randomly assign one of two different colors with equal probability. Now consider an edge $e \in E$, its two vertices have different color with probability $\frac{1}{2}$. Thus, the expected number of edges that are cut by this random assignment is $\frac{|E|}{2}$. Thus, the probability that a random assignment's cut size $\geq \frac{|E|}{2}$ is positive. Thus, there is at least one cut with size $\geq \frac{|E|}{2}$.

QED.

The proof of this lemma demonstrates a so-called "probabilistic method". This is a way to prove the existence of something. To show the existence, it suffices to show that its probability is greater than 0.

Do not confuse probabilistic method with randomized algorithm. The former is a way to prove something. It doesn't have to involve an algorithm. Whereas the latter's goal is the algorithm.

But incidentally, the proof of Max-Cut lemma provides a randomized "approximation" algorithm.

Theorem 6. The algorithm described in the proof of Max-Cut lemma computes a cut with *expected* size at least $\frac{1}{2}$ of the Max-Cut.

Proof: Notice the Max-Cut size can't exceed the total number of edges.

QED.

Derandomization

There are a few standard techniques that can sometimes convert a randomized algorithm to a deterministic one. These techniques do not always work; and derandomization is a big topic. We study only one of such techniques to get a sense of derandomization. The technique we study is called "the method of conditional probabilities".

We use the randomized Max-Cut as an example.

Recall that each step of the randomized max-cut chooses a vertex v, and randomly assign a color red or black to it. Before assigning the color, the expected cut size of the algorithm's output is $\frac{|E|}{2}$. Since

$$E[cut\ size] = P(v\ is\ red) \cdot E[cut\ size|\ v\ is\ red] + P(v\ is\ black) \cdot E[cut\ size|\ v\ is\ black)$$
$$= \frac{1}{2} \cdot E[cut\ size|\ v\ is\ red] + \frac{1}{2} \cdot E[cut\ size|\ v\ is\ black)$$

At least one of the two possibilities will be such that $E[cut\ size|\ v\ is\ \text{xxx}] \geq \frac{|E|}{2}$.

It happens that it is easy to calculate $E[cut\ size]$ given a subset of vertices are colored: Each edge with both vertices colored contributes either 0 or 1; and each edge with at least one vertex uncolored contributes $\frac{1}{2}$ to this expected value. So, we can simply calculate the two expected values by assigning v to be red and black, respectively. Then choose the one with the larger expected value. This converts the random choice to a deterministic calculation. This process can be repeated to color all other vertices, while maintaining that

$$E[cut\ size\ |\ current\ color\ assignments] \geq \frac{|E|}{2}.$$

Two key reasons for the method of conditional probabilities work:

1. The algorithm consists of a series of small random choices. Each choice may take several possible values.
2. The conditional probability for the algorithm succeeds after making a choice can be computed, or at least compared to each other (so we can take the largest).

With these two properties, the method of conditional probabilities can be applied to do derandomization. Suppose the algorithm succeeds with a positive probability $p > 0$ before making a choice. Then $P(algorithm\ success\ |\ value\ of\ choice) \geq p > 0$ for at least one value of the choice. So, one can deterministically choose the one that provides the highest conditional probability. And this process can be continued to make all the choices deterministically.

Note that in the max-cut example expected value is used instead of the conditional probability.

Unfortunately not all randomized algorithm can be derandomized this way. For example, the randomized algorithm for min-cut can't because the conditional probability cannot be easily calculated.

Randomized Complexity Classes

Algorithm design is to design efficient algorithms for a problem. It tries to reduce the time complexity. However, gradually people realized that some problems never had any efficient algorithm, despite the tremendous efforts that people have collectively put on the specific problems. So, it would be useful to prove that certain problems do not have efficient algorithms, and save future people's algorithm design efforts on these problems. There are two types of such "hardness results":

1. Lower bounds. E.g. sorting based on comparison requires at least $O(n \log n)$ comparisons.
2. Complexity classes. E.g. Max-Cut belongs to the class NP-complete. Thus, it is unlikely that you can design a polynomial time algorithm for it. Any efforts should go around.

Notice that these hardness results are for problems but not specific algorithms. We will study more about lower bounds, complexity classes, and ways to deal with NP-hard problems later in this course. But now we study a few simple classes for randomized algorithm.

First, P (polynomial time) is the class that contains the problem that can be solved in polynomial time on a Turing machine or RAM. We delay the discussion of the computing model until later in this course – since these computing models are mostly equivalent to a general computer.

Most algorithms we've studied so far (in this course and CS240, CS341) are for problems in P. For randomized algorithm, there is a similar class RP (Randomized polynomial time).

Let us introduce a conventional way to define the complexity classes. A *decision problem* is a problem that asks whether an input satisfies a certain property. Let L denotes the language (the set of all the inputs that satisfies the property). Then a decision problem asks whether $x \in L$ for input x. People often refer to the problem by the language L it defines. And when an algorithm outputs YES for an input x, we say the algorithm accepts x.

A decision problem L belongs to RP if it has an algorithm A that runs in worst case polynomial time such polynomial time such that for any input x:

- $x \in L \Rightarrow \Pr(A(x) \text{ accepts}) \geq \frac{1}{2}$,
- $x \notin L \Rightarrow \Pr(A(x) \text{ accepts}) = 0$.

Thus, the problem in RP has a Monte-Carlo algorithm with one-sided error. When the algorithm outputs Yes, it is always Yes. But when the algorithm outputs No, there may be an error.

The threshold $\frac{1}{2}$ is arbitrary. Any constant between 0 and 1 would serve the purpose.

Similarly, we have co-RP (Complement of RP). Consider a decision problem A, its complement $\tilde{A}$ is to ask whether A gives a No answer. Thus, the correct answer to A and $\tilde{A}$ always complement each other. A problem belongs to co-RP if and only if its complement belongs to RP.

ZPP (zero-error probabilistic polynomial time) is the problems that have polynomial time Las Vegas algorithms.

Exercise: Show that $\mathrm{ZPP} = \mathrm{RP} \cap \text{co-RP}$.

PP (Probabilistic Polynomial time) contains of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x,

- $x \in L \Rightarrow \Pr(A(x) \text{ accepts}) > \frac{1}{2}$,
- $x \notin L \Rightarrow \Pr(A(x) \text{ accepts}) < \frac{1}{2}$.

PP consists of the problems that have randomized Monte Carlo algorithms making two-sided errors. Unfortunately this is a very weak condition. The two sided errors can be infinitely close to $\frac{1}{2}$ so that one cannot boost the probabilities by repeating the algorithm polynomial times.

BPP (Bounded-error Probabilistic Polynomial Time) contains of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x,

- $x \in L \Rightarrow \Pr(A(x) \text{ accepts}) \geq \frac{3}{4}$,
- $x \notin L \Rightarrow \Pr(A(x) \text{ accepts}) \leq \frac{1}{4}$.

Note that if a problem belongs to BPP, and such an algorithm is given, then we can repeat it to make the (two-sided) error probabilities to approach 0. So, BPP is fairly strong condition.

The relations of these complexity classes with other classes are not completely known. Some known relations include:

Theorem: $P \subseteq ZPP \subseteq RP \subseteq NP \subseteq PP$.

But it is not known whether these inclusions are strict. In another word, people still do not know if randomization indeed provides much power in algorithm design.

Theorem: $RP \subseteq BPP$.

But it is not known whether the inclusion is strict.

The open questions include:

1. Is $RP = co\text{-}RP$?
2. Is $BPP \subseteq NP$?

Primality Test

Primality Test: Given a positive odd number n, answers whether it is a prime.

The complement problem of primality test is therefore the composite test:

Composite Test: Given a positive odd number n, answers whether it is not a prime (a composite).

Note that these two problems can be solved by trivial algorithms in time polynomial of n. However, the input size is actually $\log n$. Therefore, a polynomial of n is exponential to the input size. We are looking for efficient algorithm whose time complexity is a polynomial of $\log n$.

First some preliminaries before we introduce the algorithm.

Let $p > 2$ be a prime number. Then Z_p is the field that has p elements $0, 1, \dots, p-1$. The addition and multiplication operations are defined as

$$a + b = (a + b) \bmod p$$

$$a \cdot b = (a \cdot b) \bmod p$$

In such a field, the square root of unity 1 has to be 1 or $p - 1$. This is because

$$x^2 - 1 \equiv 0 \pmod{p} \Leftrightarrow (x+1)(x-1) \equiv 0 \pmod{p}$$

By Fermat's little theorem, we know that if p is prime, then for any $1 < a < p$,

$$a^{p-1} \equiv 1 \pmod{p}$$

Since $p > 2$ is odd, $p - 1$ is even. Let $p - 1 = 2^s \cdot d$ for $s \geq 1$ and d being odd. We keep calculating the square root of the above equation, one of the two following situations will happen:

1. $a^{2^{r} \cdot d} \equiv -1 \pmod{p}$ for at least one of $r = 0, 1, 2, \dots, s-1$.
2. $a^{d} \equiv 1 \pmod{p}$.

Thus, if none of the two situations happen, then p is not a prime, and a is called a witness for the compositeness of p.

It can be shown (not proved here) that for any composite n, at least $\frac{3}{4}$ of the numbers a are witnesses. Therefore, by randomly choosing many of them and test, we have a high probability to answer "Composite" to a composite n. If all of the choices failed to prove n is composite, then we answer "Prime".

This is an one-sided error Monte-Carlo algorithm. The error happens when the algorithm answers “Prime”. So, Primality Test belongs to co-RP and Composite Test belongs to RP.

Some other results also showed that Primality Test belongs to RP, so it belongs to $\mathrm{ZPP} = \mathrm{RP} \cap$ co-RP.

Remark:

- The algorithm we outlined here is called the Miller–Rabin primality test.
- People have long believed that Primality Test has a deterministic polynomial time algorithm. Such algorithms existed, but their correctness was based on some unproven mathematical conjectures.
- A recent result AKS primality test (also known as Agrawal–Kayal–Saxena primality test) provided a proved deterministic algorithm. So, Primality Test also belongs to P.
- Since P=co-P, composite test also belongs to P.
- This does not mean that you can use the algorithm to factorize a composite though – to prove a number is composite, one does not have to factorize it.
- Factorization can be solved in polynomial time on a quantum computer. One does not know whether it is in P or not.

We will learn more randomized algorithms in this course.