CSCI 1590
Intro to Computational Complexity
Randomized Computation

John E. Savage
Brown University
April 15, 2009
Summary

1. Randomized Algorithms
2. Average-Case Efficiency
3. Correctness with Bounded Probability
4. Generality of Definitions
As highlighted last time, we have seen a wide range of computational models. Different models highlight different computational resources, although often these models relate. Examples:

- Turing machines vs. RAM
- circuits vs. VLSI
- PRAM vs. Families of circuits
- families of circuits vs. Turing machines

The complexity classes defined so far characterize computations with deterministic outcomes. That is, with deterministic Turing machines (DTMs), Circuits, PRAM, even NDTMs, the same input always yields the same output.

Is it realistic to consider a model of computation with probabilistic outcomes? What if a Turing machine has access to random bits. There do appear to be sources of true randomness in the universe. Randomness can also be used to model uncertainty.
Randomness: A new resource

- As highlighted last time, we have seen a wide range of computational models. Different models highlight different computational resources, although often these models relate. Examples:
  - Turing machines vs. RAM
  - circuits vs. VLSI
  - PRAM vs. Families of circuits
  - families of circuits vs. Turing machines

- The complexity classes defined so far characterize computations with deterministic outcomes. That is, with deterministic Turing machines (DTMs), Circuits, PRAM, even NDTMs, the same input always yields the same output.
Randomness: A new resource

- As highlighted last time, we have seen a wide range of computational models. Different models highlight different computational resources, although often these models relate. Examples:
  - Turing machines vs. RAM
  - circuits vs. VLSI
  - PRAM vs. Families of circuits
  - families of circuits vs. Turing machines

- The complexity classes defined so far characterize computations with deterministic outcomes. That is, with deterministic Turing machines (DTMs), Circuits, PRAM, even NDTMs, the same input always yields the same output.

- Is it realistic to consider a model of computation with probabilistic outcomes? What if a Turing machine has access to random bits.
Randomness: A new resource

- As highlighted last time, we have seen a wide range of computational models. Different models highlight different computational resources, although often these models relate. Examples:
  - Turing machines vs. RAM
  - circuits vs. VLSI
  - PRAM vs. Families of circuits
  - families of circuits vs. Turing machines

- The complexity classes defined so far characterize computations with deterministic outcomes. That is, with deterministic Turing machines (DTMs), Circuits, PRAM, even NDTMs, the same input always yields the same output.

- Is it realistic to consider a model of computation with probabilistic outcomes? What if a Turing machine has access to random bits.

- There do appear to be sources of true randomness in the universe.

- Randomness can also be used to model uncertainty.
Randomness and Turing Machines

- We wish to consider what can be computed by a **probabilistic Turing machine** (PTM), a Turing machine in which state transitions occur probabilistically.
We wish to consider what can be computed by a **probabilistic Turing machine** (PTM), a Turing machine in which state transitions occur probabilistically.

To make this precise, let a PTM be a DTM with access to a second tape containing only random bits. As it computes, it can read and write from its input/work tape or read random bits from its second tape with random bits.
Randomness and Turing Machines

- We wish to consider what can be computed by a **probabilistic Turing machine** (PTM), a Turing machine in which state transitions occur probabilistically.

- To make this precise, let a PTM be a DTM with access to a second tape containing only random bits. As it computes, it can read and write from its input/work tape or read random bits from its second tape with random bits.

- This PTM model is similar to an NDTM which can be viewed as a DTM with access to a properly chosen certificate. It also allows us to quantify how much “randomness” a given computation uses.
We wish to consider what can be computed by a **probabilistic Turing machine** (PTM), a Turing machine in which state transitions occur probabilistically.

To make this precise, let a PTM be a DTM with access to a second tape containing only random bits. As it computes, it can read and write from its input/work tape or read random bits from its second tape with random bits.

This PTM model is similar to an NDTM which can be viewed as a DTM with access to a properly chosen certificate. It also allows us to quantify how much “randomness” a given computation uses.

What should our goal be when computing with a PTM?
We wish to consider what can be computed by a **probabilistic Turing machine** (PTM), a Turing machine in which state transitions occur probabilistically.

To make this precise, let a PTM be a DTM with access to a second tape containing only random bits. As it computes, it can read and write from its input/work tape or read random bits from its second tape with random bits.

This PTM model is similar to an NDTM which can be viewed as a DTM with access to a properly chosen certificate. It also allows us to quantify how much “randomness” a given computation uses.

What should our goal be when computing with a PTM?

- To always output the correct output, but to be efficient (in time or space) on average?
- To always execute efficiently, but to be correct with bounded probability?
Average-Case Efficiency

Definition

Let $\text{ZPTIME}(T(n))$ be the class of all languages, $L$, for which there is a PTM, $M$ that runs in expected time $O(T(n))$ and that outputs 1 when $x \in L$ and 0 otherwise. Let $\text{ZPP} = \bigcup_{c \geq 1} \text{ZPTIME}(n^c)$. 

What is meant by “expected time $O(T(n))$”?

On a given input, $x$, the expected runtime is the number of steps until $M$ halts, averaged over all random strings. Typically it suffices to average over random strings of some bounded length. $M$ runs in expected time $f(n)$ if $f(n)$ is the maximum of expected runtimes over all $x$ of length $n$. $M$ runs in expected time $O(T(n))$ if $f(n) = O(T(n))$. 

In practice, randomness is often useful to ensure good average case performance (for example, routing data on a hypercube).
Average-Case Efficiency

Definition

Let $\text{ZPTIME}(T(n))$ be the class of all languages, $L$, for which there is a
PTM, $M$ that runs in expected time $O(T(n))$ and that outputs 1 when 
$x \in L$ and 0 otherwise. Let $\text{ZPP} = \bigcup_{c \geq 1} \text{ZPTIME}(n^c)$.

- What is meant by “expected time $O(T(n))$”?
  - On a given input, $x$, the expected runtime is the number of steps until 
    $M$ halts, averaged over all random strings. Typically it suffices to 
    average over random strings of some bounded length.
  - $M$ runs in expected time $f(n)$ if $f(n)$ is the maximum of expected 
    runtimes over all $x$ of length $n$.
  - $M$ runs in expected time $O(T(n))$ if $f(n) = O(T(n))$.
Average-Case Efficiency

**Definition**

Let $\text{ZPTIME}(T(n))$ be the class of all languages, $L$, for which there is a PTM, $M$ that runs in expected time $O(T(n))$ and that outputs 1 when $x \in L$ and 0 otherwise. Let $\text{ZPP} = \bigcup_{c \geq 1} \text{ZPTIME}(n^c)$.

- What is meant by “expected time $O(T(n))$”?
  - On a given input, $x$, the expected runtime is the number of steps until $M$ halts, averaged over all random strings. Typically it suffices to average over random strings of some bounded length.
  - $M$ runs in expected time $f(n)$ if $f(n)$ is the maximum of expected runtimes over all $x$ of length $n$.
  - $M$ runs in expected time $O(T(n))$ if $f(n) = O(T(n))$.

- In practice, randomness is often useful to ensure good average case performance (for example, routing data on a hypercube).
An Example of Expected Runtime Analysis

- Comparison-based sorting of $n$ integers is known to be $\Omega(n \log n)$. 

A merge sort runs in time $O(n \log n)$. Quicksort runs in time $O(n^2)$ but often works well in practice. Why?

In quicksort a random element of the list is selected as a "pivot". The remaining elements are compared to the pivot and partitioned into two groups. The algorithm then recurses on each group. On average only $O(n \log n)$ comparisons are required. To see why, let $X_{i,j}$ be 1 if elements $i$ and $j$ are compared at some point during the algorithms execution, and 0 otherwise. $X_{i,j} = 1$ if and only if no element between $i$ and $j$ is selected as a pivot before $i$ or $j$. Since elements are selected as pivots in a random order, $P[X_{i,j} = 1] = \frac{2}{|i-j|+1}$. Since elements are only selected as pivots once, and no two elements are compared twice, the expected number of comparisons is $E[\sum_{i<j} X_{i,j}] = \sum_{i<j} E[X_{i,j}] = \sum_{i<j} 2\left(\frac{1}{|i-j|+1}\right) \leq n(1 + \frac{1}{2} + \ldots + \frac{1}{n-2})$. 

John E. Savage (Brown University)  
CSCI 1590 Intro to Computational Complexity  
April 15, 2009 6 / 11
Comparison-based sorting of \( n \) integers is known to be \( \Omega(n \log n) \).

A merge sort runs in time \( O(n \log n) \). Quicksort runs in time \( O(n^2) \) but often works well in practice. Why?
An Example of Expected Runtime Analysis

- Comparison-based sorting of \( n \) integers is known to be \( \Omega(n \log n) \).
- A merge sort runs in time \( O(n \log n) \). Quicksort runs in time \( O(n^2) \) but often works well in practice. Why?
- In quicksort a random element of the list is selected as a “pivot”. The remaining elements are compared to the pivot and partitioned into two groups. The algorithm then recurses on each group.
An Example of Expected Runtime Analysis

- Comparison-based sorting of \( n \) integers is known to be \( \Omega(n \log n) \).
- A merge sort runs in time \( O(n \log n) \). Quicksort runs in time \( O(n^2) \) but often works well in practice. Why?
- In quicksort a random element of the list is selected as a “pivot”. The remaining elements are compared to the pivot and partitioned into two groups. The algorithm then recurses on each group.
- On average only \( O(n \log n) \) comparisons are required.
An Example of Expected Runtime Analysis

- Comparison-based sorting of \( n \) integers is known to be \( \Omega(n \log n) \).
- A merge sort runs in time \( O(n \log n) \). Quicksort runs in time \( O(n^2) \) but often works well in practice. Why?
- In quicksort a random element of the list is selected as a “pivot”. The remaining elements are compared to the pivot and partitioned into two groups. The algorithm then recurses on each group.
- On average only \( O(n \log n) \) comparisons are required.
  - To see why, let \( X_{i,j} \) be 1 if elements \( i \) and \( j \) are compared at some point during the algorithms execution, and 0 otherwise.
An Example of Expected Runtime Analysis

- Comparison-based sorting of $n$ integers is known to be $\Omega(n \log n)$.
- A merge sort runs in time $O(n \log n)$. Quicksort runs in time $O(n^2)$ but often works well in practice. Why?
- In quicksort a random element of the list is selected as a “pivot”. The remaining elements are compared to the pivot and partitioned into two groups. The algorithm then recurses on each group.
- On average only $O(n \log n)$ comparisons are required.
  - To see why, let $X_{i,j}$ be 1 if elements $i$ and $j$ are compared at some point during the algorithms execution, and 0 otherwise.
  - $X_{i,j} = 1$ if and only if no element between $i$ and $j$ is selected as a pivot before $i$ or $j$. Since elements are selected as pivots in a random order, $P[X_{i,j} = 1] = \frac{2}{|i-j|+1}$
An Example of Expected Runtime Analysis

- Comparison-based sorting of $n$ integers is known to be $\Omega(n \log n)$.
- A merge sort runs in time $O(n \log n)$. Quicksort runs in time $O(n^2)$ but often works well in practice. Why?
- In quicksort a random element of the list is selected as a “pivot”. The remaining elements are compared to the pivot and partitioned into two groups. The algorithm then recurses on each group.
- On average only $O(n \log n)$ comparisons are required.
  - To see why, let $X_{i,j}$ be 1 if elements $i$ and $j$ are compared at some point during the algorithms execution, and 0 otherwise.
  - $X_{i,j} = 1$ if and only if no element between $i$ and $j$ is selected as a pivot before $i$ or $j$. Since elements are selected as pivots in a random order, $P[X_{i,j} = 1] = \frac{2}{|i-j|+1}$
  - Since elements are only selected as pivots once, and no two elements are compared twice, the expected number of comparisons is $E[\sum\sum_{i<j}X_{i,j}] = \sum\sum_{i<j}E[X_{i,j}] = \sum\sum_{i<j} \frac{2}{|i-j|+1} \leq n\left(1 + \frac{1}{2} + \ldots + \frac{1}{n-2}\right)$
Correctness with high probability

Definition

Let $\text{BPTIME}(T(n))$ be the class of all languages, $L$, for which there is a PTM, $M$, that runs in time $O(T(n))$ and for all $x$

- $M$ outputs 1 with probability at least $2/3$ when $x \in L$.
- $M$ outputs 0 with probability at least $2/3$ when $x \notin L$.

Let $\text{BPP} = \bigcup_{c \geq 1} \text{BPTIME}(n^c)$. 
Correctness with high probability

**Definition**

Let $\text{BPTIME}(T(n))$ be the class of all languages, $L$, for which there is a PTM, $M$, that runs in time $O(T(n))$ and for all $x$

- $M$ outputs 1 with probability at least 2/3 when $x \in L$.
- $M$ outputs 0 with probability at least 2/3 when $x \not\in L$.

Let $\text{BPP} = \bigcup_{c \geq 1} \text{BPTIME}(n^c)$.

- As with $\text{ZPP}$, we have placed requirements on the PTM’s behavior that must hold for every input, as we range over all random strings.
Let $\text{BPTIME}(T(n))$ be the class of all languages, $L$, for which there is a PTM, $M$, that runs in time $O(T(n))$ and for all $x$

- $M$ outputs 1 with probability at least 2/3 when $x \in L$.
- $M$ outputs 0 with probability at least 2/3 when $x \notin L$.

Let $\text{BPP} = \bigcup_{c \geq 1} \text{BPTIME}(n^c)$.

- As with $\text{ZPP}$, we have placed requirements on the PTM’s behavior that must hold for every input, as we range over all random strings.
- In this definition 2/3’s acts as an arbitrary constant greater than 1/2.
One-Sided Errors

**Definition**

Let \( \text{RPTIME}(T(n)) \) be the class of all languages, \( L \), for which there is a PTM, \( M \), that runs in time \( O(T(n)) \) and for all \( x \)

- \( M \) outputs 1 with probability at least 2/3 when \( x \in L \).
- \( M \) outputs 0 with probability 1 when \( x \notin L \).

Let \( \text{RP} = \bigcup_{c \geq 1} \text{RPTIME}(n^c) \).
One-Sided Errors

Definition

Let $\text{RPTIME}(T(n))$ be the class of all languages, $L$, for which there is a PTM, $M$, that runs in time $O(T(n))$ and for all $x$

- $M$ outputs 1 with probability at least $2/3$ when $x \in L$.
- $M$ outputs 0 with probability 1 when $x \notin L$.

Let $\text{RP} = \bigcup_{c \geq 1} \text{RPTIME}(n^c)$.

- Now any errors made by $M$ are one-sided. Notice that $\text{RP} \subseteq \text{NP}$. 
One-Sided Errors

Definition

Let $\text{RPTIME}(T(n))$ be the class of all languages, $L$, for which there is a PTM, $M$, that runs in time $O(T(n))$ and for all $x$

- $M$ outputs 1 with probability at least $2/3$ when $x \in L$.
- $M$ outputs 0 with probability 1 when $x \notin L$.

Let $\text{RP} = \bigcup_{c \geq 1} \text{RPTIME}(n^c)$.

- Now any errors made by $M$ are one-sided. Notice that $\text{RP} \subseteq \text{NP}$.
- Many well-known probabilistic algorithms only exhibit a one-sided error. Two examples are primality testing and identity testing.
Identity Testing

- We wish to test a polynomial identity $p(x) = q(x)$ or a matrix identity $A = BC$, which is equivalent to $Ax = B(Cx)$ for all vectors $x$. In both cases there is a simple probabilistic approach.

- Select random integers for the $n$ elements of $x$.
- Test the identity.
- To keep values small, testing can also be done mod $y$, where $y$ is a random integer.
- Repeat as needed.

Why does this work? How do we know what ranges to use?

In a polynomial $p(x)$ of total degree at most $d$, when each $x_i$ is chosen from any finite set of integers, $S$, the probability that $p(x) \neq 0$ is at least $1 - d/|S|$. This can be shown by induction.

Our range need only be $O(S)$.

The prime number theorem states that there are approximately $n/\log(n)$ primes less than $n$. When we select $O(\log(d))$ random values of $y$, one is likely to be prime.

If two polynomials differ, they are likely to differ mod a prime.
Identity Testing

- We wish to test a polynomial identity \( p(x) = q(x) \) or a matrix identity \( A = BC \), which is equivalent to \( Ax = B(Cx) \) for all vectors \( x \). In both cases there is a simple probabilistic approach.
- Select random integers for the \( n \) elements of \( x \).
- Test the identity.
- To keep values small, testing can also be done mod \( y \), where \( y \) is a random integer.
- Repeat as needed.
Identity Testing

- We wish to test a polynomial identity \( p(x) = q(x) \) or a matrix identity \( A = BC \), which is equivalent to \( Ax = B(Cx) \) for all vectors \( x \). In both cases there is a simple probabilistic approach.
  - Select random integers for the \( n \) elements of \( x \).
  - Test the identity.
  - To keep values small, testing can also be done mod \( y \), where \( y \) is a random integer.
  - Repeat as needed.
- Why does this work? How do we know what ranges to use?
  - In a polynomial \( p(x) \) of total degree at most \( d \), when each \( x_i \) is chosen from any finite set of integers, \( S \), the probability that \( p(x) \neq 0 \) is at least \( 1 - d/|S| \). This can be shown by induction.
  - Our range need only be \( O(S) \)
We wish to test a polynomial identity \( p(x) = q(x) \) or a matrix identity \( A = BC \), which is equivalent to \( Ax = B(Cx) \) for all vectors \( x \). In both cases there is a simple probabilistic approach.

- Select random integers for the \( n \) elements of \( x \).
- Test the identity.
- To keep values small, testing can also be done mod \( y \), where \( y \) is a random integer.
- Repeat as needed.

Why does this work? How do we know what ranges to use?

- In a polynomial \( p(x) \) of total degree at most \( d \), when each \( x_i \) is chosen from any finite set of integers, \( S \), the probability that \( p(x) \neq 0 \) is at least \( 1 - d/|S| \). This can be shown by induction.
- Our range need only be \( O(S) \)
- The prime number theorem states that there are approximately \( n/\log(n) \) primes less \( n \). When we select \( O(\log(d)) \) random values of \( y \), one is likely to be prime.
- If two polynomials differ, they are likely to differ mod a prime.
We wish to determine if an integer, $N$, is prime. To do this we can repeatedly select a random integer between 1 and $N - 1$ and perform a simple test.

- Let $G_N(A) = A^{(N-1)/2} \mod N$.
- Let $\gcd(A, N)$ be the greatest common divisor of $A$ and $N$.
- Let $(\frac{N}{A}) = \prod_{i=1}^{k} G_{P_i}(A)$, where $P_1, ..., P_k$ are the prime factors of $N$ (but don’t worry, you don’t need to factor $N$ to compute $(\frac{N}{A})$!)
- If $\gcd(N, A) \neq 1$ or $(\frac{N}{A}) \neq G_N(A)$, $N$ is not prime.
Solovay-Strassen Primality Testing

We wish to determine if an integer, $N$, is prime. To do this we can repeatedly select a random integer between 1 and $N - 1$ and perform a simple test.

- Let $G_N(A) = A^{(N-1)/2} \mod N$.
- Let $\gcd(A, N)$ be the greatest common divisor of $A$ and $N$.
- Let $(\frac{N}{A}) = \prod_{i=1}^{k} G_{P_i}(A)$, where $P_1, \ldots, P_k$ are the prime factors of $N$ (but don’t worry, you don’t need to factor $N$ to compute $(\frac{N}{A})$!)
- If $\gcd(N, A) \neq 1$ or $(\frac{N}{A}) \neq G_N(A)$, $N$ is not prime.

Why does this work?

- Define $QR_N(A)$ to be 0 if $\gcd(A, N) \neq 1$, 1 if $A = B^2 \mod N$ for some $B$, and -1 otherwise. Notice that $QR_N(A) = G_N(A)$ when $N$ is an odd prime, as does $(\frac{N}{A})$.
- If $\gcd(A, N) \neq 1$, $N$ is not prime. If $\gcd(A, N) = 1$, it turns out that the probability that $(\frac{N}{A}) = G_N(A)$ is at most $1/2$.
- Fortunately $(\frac{N}{A})$, called the Jacobi symbol, can be computed in time $O(\log(N) \log(A))$. 
In our definition of **BPP** and **RP**, we have already noted that 2/3 is an arbitrary constant. In fact, it can be replaced not only with a smaller constant, but a function $1/f(n)$ where $f(n)$ is polynomial, or even exponential, in $n$. This replacement leaves both classes unchanged.
How general are our definitions?

- In our definition of \textbf{BPP} and \textbf{RP}, we have already noted that $2/3$ is an arbitrary constant. In fact, it can be replaced not only with a smaller constant, but a function $1/f(n)$ where $f(n)$ is polynomial, or even exponential, in $n$. This replacement leaves both classes unchanged.

- We have defined \textbf{BPP} and \textbf{RP} in terms of the runtime of a PTM, and not its expected runtime. Using the expected runtime again leaves both classes unchanged.
How general are our definitions?

- In our definition of **BPP** and **RP**, we have already noted that 2/3 is an arbitrary constant. In fact, it can be replaced not only with a smaller constant, but a function $1/f(n)$ where $f(n)$ is polynomial, or even exponential, in $n$. This replacement leaves both classes unchanged.

- We have defined **BPP** and **RP** in terms of the runtime of a PTM, and not its expected runtime. Using the expected runtime again leaves both classes unchanged.

- In defining **BPP**, **RP** and **ZPP** we referred to a PTM with access to random bits. Each random bit is assumed to take value 1 with probability 1/2. Replacing 1/2 with some other constant would not effect any of these three classes.