This is more material from Chapter 7 of the text.  
We saw (without proof) that there is an important application of BPP algorithms.

Consider a multivariate polynomial $P(x_1, x_2, \cdots x_n)$ of total degree $d$. Suppose we have only 'black box access' to it—we can evaluate the polynomial (say, at cost 1) for any input $a_1, \cdots a_n$, but we do not have access at the coefficients. The task is to determine whether $P()$ is the identically 0 polynomial.

Lemma 7.5 in the book shows that the following simple algorithm works: take a bunch of numbers at random, evaluate the polynomial at these places. One of them is going to be nonzero if the polynomial is not identically 0. More precisely, if we take a random element element from a set $S$ of cardinality $|S|$, we have

$$Pr[P(a_1, a_2, \cdots a_n) \neq 0] \geq 1 - d/|S|$$

We shall not prove this, but will need it later in the course.

\section{Upper bounds on BPP}

We do not know the exact complexity of $BPP$. As we saw during the last lecture, it is not impossible that $BPP = P$. While we cannot prove this, a nonuniform version of the theorem us true.
1.1 Nonuniform bounds

Theorem 1. $P_{BB} \subseteq P/poly$.

Proof. Let $L \in BPP$. That means that there is a polynomial time bounded machine $M'$ in BPP that accepts $L$ with error probability $\leq 1/3$. We saw that we can make the probability of error be as small as $1/2n + 1$ for inputs of length $n$, and still work in polynomial time. Let $M$ be such a machine. This means that $M$ has the following property: let $n \in \mathbb{N}$. Then

$$\forall x \in \{0, 1\}^n \ Pr[M(x, r) \neq L(x)] \leq 1/3$$

where $r$ is a polynomially long ($|r| = m \leq n^c$) random string. Say that such a string $r$ is good if $M(x, r) = L(x)$, and bad if $M(x, r) \neq L(x)$. Now build an $2^n \times 2^m$ Boolean matrix $A$, where the rows correspond to the $2^n$ possible input strings of length $n$, and the columns to the $2^m$ possible random strings that $M$ may use in its computation. $A$ is defined as

$A[i, j] = 1$ if the random string $j$ is good for the computation of $i$ (testing whether $i \in L$), and $A[i, j] = 0$ if $j$ is bad.

We claim that there must be a column of all 1s. (corresponding to a random string $r$ that is good for every input.)

This is true because only a $1/2n+1$ fraction of all bits is bad (because of the error probability of the $BPP$ machine.) This means that there are at most $1/2n+1 \times 2^n 2^m = 2^m/2$ bad bits (0s) in the matrix. Since there are $2^m$ columns, there must be a column of all 1s.

This means that if we use $r$ as a ‘random string’, $M$ will give the correct answer for every $x$ of length $n$. If we give the string to $M$, it will be deterministic, use a polynomial length help string, and run in polynomial time.

Of course, the string $r$ needed will depend on $n$, and we will need different strings for each $n$. Also, the proof that yielded the existence of the string was nonconstructive: we do not have an efficient algorithm to compute it.

2 Other Models of Randomness

We can also consider reliable probabilistic computations with 1-sided error.

Definition 2. $RPTIME[T(n)]$ is the class of languages accepted by probabilistic Turing machines that on inputs $x$ of length $n$
• take at most \(dT(n)\) steps (where \(d > 0\) does not depend on \(n\))

• have the following correctness guarantee:
  
  - (Completeness) if \(x \in L\) \(Pr[M(x) = L(x)] \geq 2/3\) (this is the same as \(Pr[M(x) = 1] \geq 2/3\))
  
  - (Soundness) if \(x \notin L\) \(Pr[M(x) = 1] = 0\)

**Definition 3.** \(RP = \cup_{i>0}RPTIME[n^i]\)

\(RP\) gives a stronger guarantee than \(BPP\). In particular, even if we were to define \(x \in L\) \(Pr[M(x) = L(x)] \geq a\) for any constant \(a > 0\), we would get the same class \(RPP\). As we shall see in an exercise, it is also true that we can reduce the probability of error to be exponentially small – with a much simper proof.

Since the definition of \(RP\) is not symmetric with respect to complements, it makes sense to define explicitly the class \(coRP\) of languages whose complement is in \(RP\), and to look at the class \(RP \cap coRP\).

We can define also zero error probabilistic computations.

**Definition 4.** \(ZTIME[T(n)]\) is the class of languages accepted by probabilistic Turing machines that on inputs \(x\) of length \(n\) have \(Pr[M(x) = L(x)] = 1\) and the expected runtime of \(M\) is \(T(n)\).

We also define \(ZPP = \cup_{i>0}ZTIME[n^i]\)

It is easy to see that \(RP \subseteq NP\) (and \(coRP \subseteq coNP\)).

We’ll see (homework problem) that \(ZPP = RP \cap coRP\).

### 2.1 Uniform Upper Bounds

Next class we will prove

**Theorem 5.** \(BPP \subseteq \Sigma_2^p \cap \Pi_2^p\)
3 Space Bounded Random Computations

We do not have time to look carefully at the analogues of BPP for space bounded computation. I’ll just state – without proof that one can solve the GAP problem (given graph $G$ and two vertices $s$ and $t$, is there a path from $s$ to $t$ in $G$?) for undirected graphs in $RL$, the logspace equivalent for $RP$. This is somewhat surprising, as the GAP problem (for directed graphs) is complete for $L$ and the best algorithm uses $\Theta((\log n)^2)$ space.

The randomized algorithm is simple: take a random walk of polynomial length starting at $s$ and see if the walk ever reaches $t$. The algorithm only has to keep track of the current vertex $v$, $t$, the next vertex, and a counter (of length $O(\log n)$) to make sure that the process halts after polynomially many steps.

The proof of correctness is not difficult, but is best done via linear algebra... so we won’t do it.

4 Complete problems?

Don’t know of any for $BPP$...

Or for RP. The problem is that unlike the classes we have seen previously – $P$, $L$, $NP$, etc.– $BPP$ cannot be described easily by a ‘syntactic’ machine. By ‘syntactic’ I mean that there is a way of listing machines that are guaranteed to be in the set, and that all functions are on the list. (There was a midterm problem asking you to do this for $P$). The property of obeying the completeness and soundness criteria is something that is done at runtime–there does not seem to be a mechanism to produce machines that automatically satisfy these conditions.