Lecture 11: Efficient Algorithms
In today’s lecture capital alphabets, for example, $X$, represents a natural number.

Further, the number of bits needed to present the number $X$ is denoted by the corresponding small number $x$. 
Note that the smallest integer $X$ that requires $n$ bits for binary representation has the binary representation $10\cdots0$. This represents the number $X = 2^{n-1}$.

Note that the largest integer $X$ that can be expressed using $n$ bits has binary representation $1\cdots1$. This represents the number $X = 2^n - 1$.

From these two observations, we can conclude that the number of bits needed to represent any number $X$ is given by $x = \left\lceil \lg(X + 1) \right\rceil$.

Intuitive Summary: The number $X$ requires $x = \lg X$ bits for its representation.
An efficient algorithm is an algorithm whose running time is polynomial in the size of the input.

For example, suppose an algorithm takes as input a prime $P$ that needs $p = 1000$ bits to represent it. Note that the prime $P$ is at least $2^{1000-1} = 2^{999}$, which is humongous (more than the number of atoms in the universe). Our algorithm’s running time should be polynomial in $p = 1000$, rather than the number $P \geq 2^{999}$.

We shall assume that all inputs are already provided in the binary representation.
Suppose we are given two numbers $A$ and $B$. Our objective is to generate the binary representation of the sum of these two numbers.

Note that $A$ needs $a = \lceil \lg(A + 1) \rceil$ and $B$ needs $b = \lceil \lg(B + 1) \rceil$ bits for representation.
**Naive Attempt.**

Add($A, B$):
- $\text{sum} = A$
- For $i = 1$ to $B$:
  - $\text{sum}+ = 1$
- Return $\text{sum}$

Note that the inner loop runs $B$ times, which is at least $2^{b-1}$, i.e., exponential in the input size. So, this algorithm is inefficient.
Efficient Addition Algorithm.

\[
\text{Add}(A, B):
\begin{align*}
&\cdot c = \max\{a, b\}, \text{ carry } = 0 \\
&\cdot \text{ For } i = 0 \text{ to } c - 1:
\begin{align*}
&\quad \cdot C_i = A_i + B_i + \text{ carry} \\
&\quad \cdot \text{ If } C_i \geq 2:
\begin{align*}
&\quad \quad \cdot \text{ carry } = 1 \\
&\quad \quad \cdot C_i = C_i \% 2
\end{align*}
\end{align*}
\end{align*}
\begin{align*}
&\quad \cdot \text{ Else: carry } = 0 \\
&\quad \cdot \text{ If carry } == 1:
\begin{align*}
&\quad \quad \cdot c+ = 1 \\
&\quad \quad \cdot C_{c-1} = 1
\end{align*}
\end{align*}
\begin{align*}
&\cdot \text{ Return } C_{c-1}C_{c-2} \ldots C_1C_0
\end{align*}
\]
The running time of this algorithm is $O(a + b)$, where $a = \log A$ and $b = \log B$. This algorithm is efficient!
Suppose we are given two numbers $A$ and $B$. Our objective is to generate the binary representation of the product of these two numbers.

Our algorithm should have running time polynomial in $a = \lceil \log(A + 1) \rceil$ and $b = \lceil \log(B + 1) \rceil$.
Naive Attempt.

\[ \text{Multiply}(A, B): \]
- \( \text{product} = 1 \)
- \( \text{For } i = 1 \text{ to } B: \)
  - \( \text{product} += A \)
- \( \text{Return } \text{product} \)

Note that the inner loop runs \( B \) times, which is at least \( 2^{b-1} \), i.e., exponential in the input size. So, this algorithm is inefficient.
Efficient Addition Algorithm.

Multiply\((A, B)\):
\[
\begin{align*}
&\text{to\_add} = A \\
&\text{remains} = B \\
&\text{product} = 0 \\
\text{While remains} > 0: \\
&\quad \text{If remains}\&1 = 1: \text{product} + = \text{to\_add} \\
&\quad \text{to\_add} = \text{to\_add} \ll 1 \\
&\quad \text{remains} = \text{remains} \gg 1 \\
\end{align*}
\]

Return product

The running time of this algorithm is \(O((a + b)^2)\), where \(a = \log A\) and \(b = \log B\). This algorithm is efficient!
Multiplication IV

- **Additional Reading.** Read Fast Fourier Transform for even faster multiplication algorithms!
Students are encouraged to write the pseudocode of an efficient division algorithm that takes as input integers $A$ and $B$ and outputs integers $M$ and $R$ such that

1. $B = M \cdot A + R$, and
2. $R \in \{0, \ldots, A - 1\}$
Our objective is to find the greatest common divisor \( G \) of two input integers \( A \) and \( B \).

Note that if we iterate over all integers \( \{1, \ldots, A\} \) to find the largest integer that divides \( B \), then this algorithm has a loop that runs \( A \) times, that is, it is exponential in the input length.

So, we use Euclid’s GCD algorithm. Let \( R \) be the remainder of dividing \( B \) by \( A \). If \( R = 0 \), then \( A \) is the GCD of \( A \) and \( B \). Otherwise, it recursively returns the gcd\((R, A)\). This algorithm is based on the observation that

\[
gcd(A, B) = gcd(R, A)
\]

Students are encouraged to prove this statement.
Euclid’s GCD Algorithm.

\[
\text{GCD}(A, B) \\
\begin{align*}
& \quad R = B \% A \\
\text{While } & \quad R > 0 : \\
& \quad B = A \\
& \quad A = R \\
& \quad R = B \% A \\
\end{align*} \\
\text{Return } & \quad A
\]

Exercise. Prove that this is an efficient algorithm.
The following code generates a random number in the range 
$[2^{n-1}, 2^n - 1]$

```
Random(n):
  C = 1
  For i = 1 to (n - 1):
    r ← {0, 1}
    C = (C ≪ 1) | r
```

It is easy to see that this is an efficient algorithm
Assume that there exists an efficient algorithm Is_Prime(N) that tests whether the integer N is a prime or not. In the future, we shall see one such algorithm.

Consider the following code

```python
Prime(n):
  While true :
    P = Random(n)
    If Is_Prime(P) : Return P
```

The efficiency of the above algorithm depends on the number of times the while-loop runs, which depends on the number of primes in the range \([2^{n-1}, 2^n - 1]\)
Generate a Random $n$-bit Prime II

- We shall rely on the density of prime numbers to understand the running time of the algorithm mentioned above.

**Theorem (Prime Number Theorem)**

There are (roughly) $\frac{N}{\log N}$ prime numbers $< N$

- So, there are roughly $\frac{2^n}{n}$ prime numbers $< 2^n$. Similarly, there are roughly $\frac{2^{n-1}}{n-1}$ prime numbers $< 2^{n-1}$. So, in the range $[2^{n-1}, 2^n - 1]$, the number of primes is (roughly)

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

- The range $[2^{n-1}, 2^n - 1]$ has a total of $2^{n-1}$ numbers.
So, the probability that a random number picked from this range is a prime number is (roughly)

\[
\frac{2^{n-1} \cdot \frac{1}{n}}{2^{n-1}} = \frac{1}{n}
\]

Intuitively, if we run the inner-loop \( n \) times, then we expect to encounter one prime number. We shall make this more formal in the next class.

I want to emphasize that if the density of the primes was not \( 1/\text{poly}(n) \), then the algorithm presented above will not be efficient. We are extremely fortunate that primes are so dense!