

# Lecture 10: Efficient Algorithms

# Notation Convention

- 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  $x$

# Length of Representation

- Note that the smallest integer  $X$  that requires  $n$  bits for binary representation has the binary representation  $1 \overbrace{0 \cdots 0}^{(n-1)\text{-times}}$ . This represents the number  $X = 2^{n-1}$ .
- Note that the largest integer  $X$  that can be expressed using  $n$  bits has binary representation  $\overbrace{1 \cdots 1}^{n\text{-times}}$ . 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 give by  $x = \lceil \lg(X + 1) \rceil$
- Intuitive Summary: The number  $X$  requires  $x = \lceil \lg X \rceil$  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 number  $A$  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.**

Add( $A, B$ ):

- $c = \max\{a, b\}$ , carry = 0
- For  $i = 0$  to  $c - 1$ :
  - $C_i = A_i + B_i + \text{carry}$
  - If  $C_i \geq 2$ :
    - carry = 1
    - $C_i = C_i \% 2$
  - Else: carry = 0
- If carry == 1:
  - $c++$
  - $C_{c-1} = 1$
- Return  $C_{c-1}C_{c-2} \dots C_1C_0$

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



# Multiplication I

- Suppose we are given two number  $A$  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  $a = \lceil \lg(A + 1) \rceil$  and  $b = \lceil \lg(B + 1) \rceil$

- **Naive Attempt.**

Multiply( $A, B$ ):

- product = 1
  - For  $i = 1$  to  $B$ :
    - product+ =  $A$
  - Return 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( $A, B$ ):

- $to\_add = A$
  - $remains = B$
  - $product = 0$
  - While  $remains > 0$ :
    - If  $remains \& 1 = 1$ :  $product += to\_add$
    - $to\_add = to\_add \ll 1$
    - $remains = remains \gg 1$
  - 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!

- **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  $R \in \{0, \dots, A - 1\}$

# Finding Greatest Common Divisor I

- 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  $\{1, \dots, A\}$  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  $\text{gcd}(R, A)$ . This algorithm is based on the observation that

$$\text{gcd}(A, B) = \text{gcd}(R, A)$$

Students are encouraged to prove this statement.

- **Euclid's GCD Algorithm.**

GCD( $A, B$ )

- $R = B \% A$
- While  $R > 0$  :
  - $B = A$
  - $A = R$
  - $R = B \% A$
- Return  $A$

- **Exercise.** Prove that this is an efficient algorithm.

# Generate $n$ -bit Random Number

- The following code generates a random number in the range  $[2^{n-1}, 2^n - 1]$

Random( $n$ ):

- $C = 1$
  - For  $i = 1$  to  $(n - 1)$ :
    - $r \stackrel{s}{\leftarrow} \{0, 1\}$
    - $C = (C \ll 1) | r$
- It is easy to see that this is an efficient algorithm



# Generate a Random $n$ -bit Prime I

- **Assume** that there exists an efficient algorithm  $\text{Is\_Prime}(N)$  that tests whether the integer  $N$  is a prime or not. In the future, we shall see one such algorithm.
- Consider the following code

```
Prime( $n$ ):
```

- While true :
  - $P = \text{Random}(n)$
  - If  $\text{Is\_Prime}(P)$  : 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)  $N/\log N$  prime numbers  $< N$*

- So, there are roughly  $2^n/n$  prime numbers  $< 2^n$ . Similarly, there are roughly  $2^{n-1}/(n-1)$  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.

## Generate a Random $n$ -bit Prime III

- 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!