#### Lecture 11: Efficient Algorithms

Efficient Algorithms

▲御 ▶ ▲ 臣 ▶ ▲

B> B

- 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

(4 同) ト ( ヨ ) ( ヨ )

• Note that the smallest integer X that requires n bits for binary (n-1)-times

representation has the binary representation 1  $\overbrace{0\cdots0}^{}$  . This represents the number  $X=2^{n-1}$ .

- Note that the largest integer X that can be expressed using n n-times
   bits has binary representation 1...1. This represents the number X = 2<sup>n</sup> 1.
- From these two observations, we can conclude that the number of bits needed to represent any number X is give by x = [lg(X + 1)]
- Intuitive Summary: The number X requires  $x = \lg X$  bits for its representation

(日) (日) (日)

- 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  $P \ge 2^{999}$ .
- We shall assume that all inputs are already provided in the binary representation

▲御▶ ★ 国▶ ★ 国▶

- 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

▲御▶ ▲ 国▶ ▲ 国▶

# Addition II

• Naive Attempt.

Add(A, B): • sum = A • For *i* = 1 to B: • sum+ = 1 • Return sum

Note that the inner loop runs B times, which is at least 2<sup>b-1</sup>, i.e., exponential in the input size. So, this algorithm is inefficient.

Efficient Algorithms

・日・・ヨ・・ヨ・

### Addition III

• Efficient Addition Algorithm.

```
Add(A, B):
           • c = \max\{a, b\}, carry = 0
            • For i = 0 to c - 1:
                          • C_i = A_i + B_i + carry
                          • If C_i \ge 2:
                                         • carry = 1
                                         • C_i = C_i \% 2
                          • Else: carry = 0
            • If carry == 1:
                          • c + = 1
                          • C_{c-1} = 1
           • Return C_{c-1}C_{c-2}...C_1C_0
```

(4 同 ) ( ヨ ) ( ヨ )

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

・ロト ・ 日 ・ ・ ヨ ・ ・ ヨ ・ ・

э

- 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  $a = \lfloor \lg(A+1) \rfloor$  and  $b = \lfloor \lg(B+1) \rfloor$

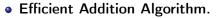
・ 同 ト ・ ヨ ト ・ ヨ ト

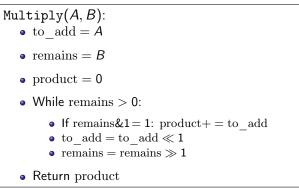
# Multiplication II

- 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<sup>b-1</sup>, i.e., exponential in the input size. So, this algorithm is inefficient.

・ 同 ト ・ ヨ ト ・ ヨ ト

## Multiplication III





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

▲御▶ ▲注▶ ▲注▶

• Additional Reading. Read Fast Fourier Transform for even faster multiplication algorithms!

・ロト ・ 日 ・ ・ ヨ ・ ・ ヨ ・

э

• 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

$$B = M \cdot A + R, \text{ and}$$

$$R \in \{0,\ldots,A-1\}$$

- 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.

・ 同 ト ・ ヨ ト ・ ヨ ト

Finding Greatest Common Divisor II

• 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.

Efficient Algorithms

・日・・ヨ・・ヨ・

• The following code generates a random number in the range  $\left\lceil 2^{n-1},2^n-1\right\rceil$ 

Random(n): • C = 1• For i = 1 to (n - 1): •  $r \stackrel{\$}{\leftarrow} \{0, 1\}$ •  $C = (C \ll 1) \mid r$ 

• It is easy to see that this is an efficient algorithm

Efficient Algorithms

▲御▶ ▲注▶ ▲注▶

- 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

```
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  $[2^{n-1}, 2^n - 1]$ 

(日本) (日本) (日本)

#### 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  $\left[2^{n-1},2^n-1
ight]$  has a total of  $2^{n-1}$  numbers.

Efficient Algorithms

- ( 同 ) - ( 回 ) - ( 回 ) - ( 回 )

• 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/poly(n), then the algorithm presented above will not be efficient. We are extremely fortunate that primes are so dense!

< 同 > < 回 > < 回 > <