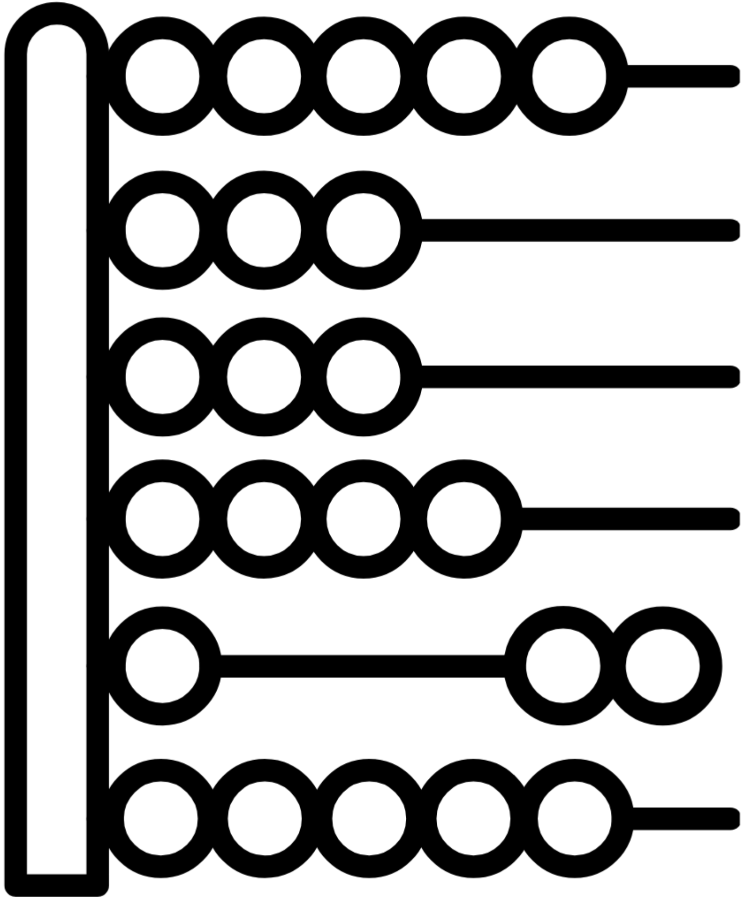
Numerics II

A Deeper Dive into CP Math



Lecture Objectives

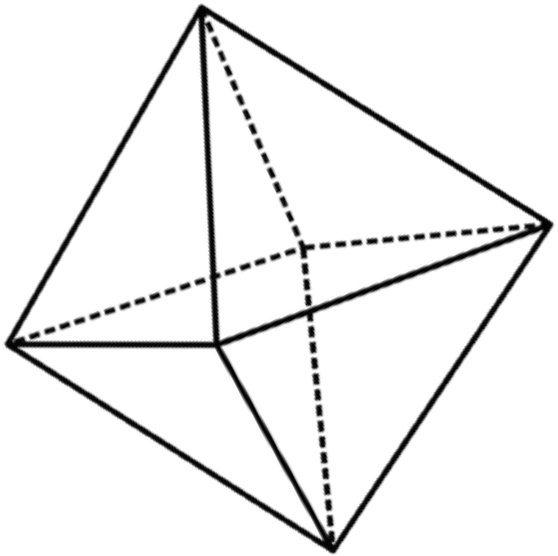
- Understanding the principles of hashing, its applications, and how hash collisions occur and are managed.
- Exploring advanced concepts in modular arithmetic in greater depth.
- Developing combinatorial reasoning skills for solving problems related to arrangements, selections, and distributions.
- Understanding Extended Euclidean Algorithm.

Lecture Outline

- Prerequisites
- Hashing and Hash Collisions
- Modular Arithmetic
- Combinatorics and Probability
- Extended Euclidean Algorithm
- Quote of the Day

Prerequisites

- Math I
  - Basic Understanding of Modular Arithmetic
  - Euclidean Algorithm
- Bit Manipulation
- Time and Space Complexity Analysis



Hashing

What is Hashing?

- Hashing is a process that converts **input data** of **any size** into a **fixed-size** string of characters/letters known as **hash value**.
- Examples:
  - Password Storage
  - Python's built-in hash function
- What is the purpose then?

What is Hashing? - Hash Functions

- In hashing, keys of **arbitrary size** are converted into keys of **fixed-size** by using **hash function.**
- What is a hash function? A **good** hash function has the following properties
    - Deterministic
    - Fixed-Size Output
    - Efficient Computation
    - Preimage Resistant
    - Second Preimage Resistance
    - Avalanche Effect
    - Uniform Distribution
    - Collision-Resistant

Hashing

- Hashing is implemented in two steps:
  - **Hash Generation:** A hash function is used to generate a **fixed-size integer or string** for a particular input
  - **Storage:** The generated hash is stored in a **large table** as a key along with the input as the value

Hash Collision

- Let `hash` be a hashing function, and let x1 and x2 be two **different** inputs. A hash collision happens when

```
hash(x1) = hash(x2)
```

- Example: if `hash = lambda x: sum(ord(c) for c in x),` then `hash('abc') = hash('cab').`

Hash Collision

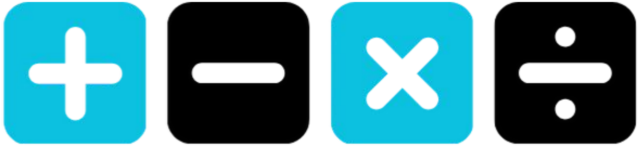
- How can we handle hashing collisions?
- Python uses hashing for basic data structures like **sets** and **dictionaries**. What are some implications of hashing collisions in competitive programming?

Modular Hashing

- Modular Hashing is mapping a key `k` into one of `m` slots by taking the remainder of `k` divided by `m`.

$$hash(k) = k \% m$$

- Modular Hashing is a **too simple** to be practical for real world applications. Which properties of a good hashing function does it satisfy and fail?



Modular Arithmetic

A2SV

A Recap on Modular Arithmetic

- Also called Clock Arithmetic
  - `(a + b) % m = (a % m + b % m) % m`
  - `(a - b) % m = (a % m - b % m) % m`
  - `(a * b) % m = (a % m * b % m) % m`
  - If `(a - b) % m = 0`, then `a % m = b % m`
  - Division is a bit more complicated.

A Recap on Modular Arithmetic

- Mathematically, we can interpret two numbers that have the same remainder as the same number and have a working algebra. This is called **Modular Congruence.**

- For example: 17 is congruent to 8 modulo 9.
- We say `17 % 9 = 8 % 9` or `17 = 8 mod 9`

Modular Division

- First of all, not all division is well-defined under modular arithmetic.
    - Consider the fraction 1 / 2 under modulo 6.
    - We know `2 * (1 / 2) = 1 mod 6.`
    - `However . . .`
        - `2 * 0 = 0`
        - `2 * 1 = 2`
        - `2 * 2 = 4`
        - `2 * 3 = 0`
        - `2 * 4 = 2`
        - `2 * 5 = 4`
    - Therefore `1 / 2` is not defined under modulo 6.

Modular Division

- For a division to be well-defined in modular arithmetic, **the divisor has to be relatively prime with the modulo.**
- This is also a sufficient condition for the division to be well-defined.

Modular Division - Naive Way

- If `a` is relatively prime with `m`, we can find the `(1/a) % m` by iterating a variable `inv_a` from `0` to `m - 1` and checking if `(inv_a * a) % m == 1`. Why?

Modular Division - Fermat's Way

- When `m` is prime, we have Fermat's little theorem:

```
1/a = a**(m - 2) mod m
```

Modular Division - Euler's Way (generalization)

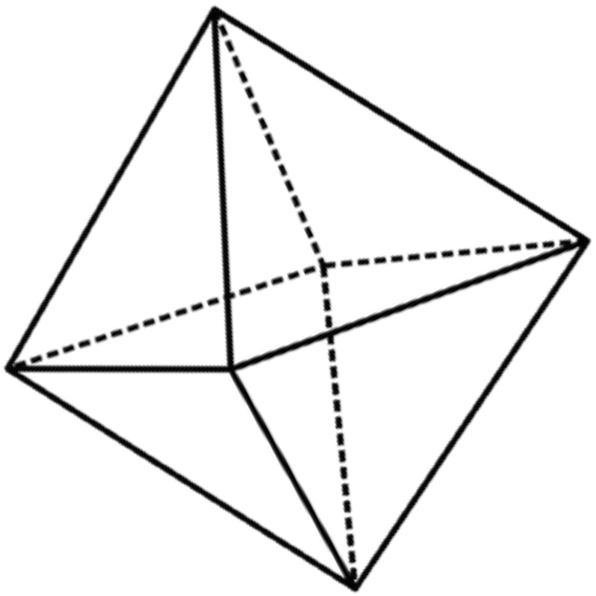
- If `m` is not a prime, there is still a way. We can use the following fact in number theory as follows:

```
1/a = a**(phi(m) - 1) mod m
```

- Where `phi(n)` is the Euler's totient function.

Proof

Implementation of Euler's totient function



Binary Exponentiation

Binary Exponentiation

- Binary exponentiation (also known as exponentiation by squaring) is a trick which allows to calculate `a**n` using only `O(log n)` multiplications.

- Let's, calculate `3**(13).`

Binary Exponentiation

- The idea of binary exponentiation is, that we split the work using the binary representation of the exponent.

$$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$$

- In general, since the number `n` has exactly `log n + 1` digits in base 2, we only need to perform `O(log n)` multiplications, if we know the powers **`a**0, a**1, a**2, a**4`** `etc.`

Binary Exponentiation

- Luckily, this is very easy. Since an element in the sequence is just the square of the previous element.

$$
\begin{aligned}
3^1 &= 3 \\
3^2 &= \left(3^1\right)^2 = 3^2 = 9 \\
3^4 &= \left(3^2\right)^2 = 9^2 = 81 \\
3^8 &= \left(3^4\right)^2 = 81^2 = 6561
\end{aligned}
$$

Binary Exponentiation

- Implementation Steps

    1. Initialize three variables:
        a. `result` to `1`
        b. `base` to the given base
        c. `exponent` to the given exponent

Binary Exponentiation

- Implementation Steps

2. While the exponent is greater than 0:
   a. If the **least significant bit (LSB)** of the exponent is 1, multiply result by base.
   b. Square the base.
   c. Right-shift exponent by 1.

3. Return result as the final result.

Binary Exponentiation - Iterative

```python
def binary_exponentiation(base, exponent):
    result = 1
    while exponent > 0:
        if exponent & 1:
            result *= base
        base *= base
        exponent >>= 1
    return result
```

Modular Arithmetic - Template

```python
class ModularArithmetic:
    def add(self, a, b, p):
        return ((a % p) + (b % p)) % p

    def subtract(self, a, b, p):
        return ((a % p) - (b % p)) % p

    def multiply(self, a, b, p):
        return ((a % p) * (b % p)) % p
```

Modular Arithmetic - Template

```python
def binary_exponentiation(self, base, exponent, p):
        result = 1
        while exponent > 0:
            if exponent  & 1:
                result = self.multiply(base, result, p)
            base = self.multiply(base, base, p)
            exponent >>= 1
        return result

    def inverse(self, a, p):
        return self.binary_exponentiation(a, p - 2, p)

    def division(self, a, b, p):
        return self.multiply(a, self.inverse(b, p), p)
```

Combinatorics

“Combinatorics is the art of counting without counting." - Andrzej Schinzel

The Addition Rule

- Suppose two experiments are to be performed.
  - If experiment `E1` has `n1` possible outcomes
  - And experiment `E2` has `n2` possible outcomes,
- then, when ***performed independently***, the total possible outcomes is `n1 + n2`.

The Addition Rule

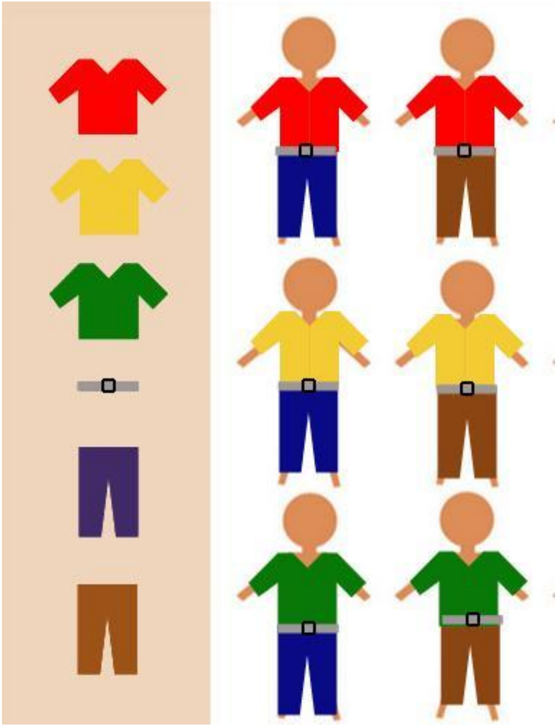
- Example:
  - Suppose you have 6 roads and 3 railways from Addis Ababa to Berland. How many possible ways do you have to go Berland?
  - Number of ways = 6 roads + 3 railways = 9.

The Multiplication Rule

- Suppose that two experiments are to be performed.
    - If there are `m` outcomes in experiment `E1`.
    - And for each outcome of experiment `E1`, there are `n` outcomes in experiment `E2`.
- Then, the total possible outcomes of **running the experiments back to back** is `m * n`.

The Multiplication Rule

- Example:



Example: Vowels of All Substrings

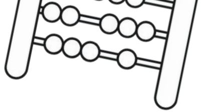
- Given a string word, return the sum of the number of vowels ('a', 'e', 'i', 'o', and 'u') in every substring of word.
- Example:

```
Input: “aba”

Output: 6

Substrings: “a”, “ab”, “aba”, “ba”, “a”

                 1 + 1 + 2 + 1 + 1 = 6
```

Example: Vowels of All Substrings

d b a c c d

- The only vowel we have is ‘a’. How do we count the number of substrings ‘a’ appears in?



Basic rules of counting

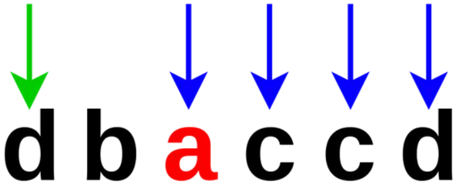
d b a c c d

The substring should start from ‘d’, ‘b’, ‘a’ inorder to include ‘a’. In our case, we have 3 possible starting points.

Basic principle of counting

- For any given index `i`, there are `i + 1` possible starting indices that allow us to include the character at index `i` .

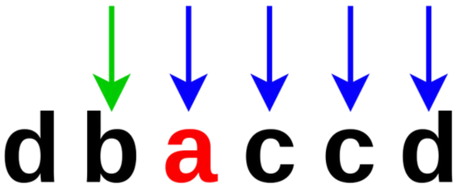
Basic rule of counting



dba
dbac
dbacc
dbaccd

If we start from ‘d’ we have 4 possible ending indices. This means we can form 4 substrings starting from ‘d’.

Basic rule of counting



ba
bac
bacc
baccd

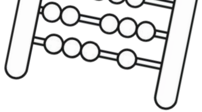
Starting from ‘b’ we can also form 4 substrings.

Basic rule of counting

d b a c c d

a
ac
acc
accd

Finally starting from ‘a’ we have 4 possible substrings.



Basic rule of counting

- In general,
    - For any vowel at index `i` (0-indexed), there are `i + 1` possible starting indices.
    - For each of these starting indices `j`, there are `n - i` possible ending indices.

Which counting principle is applicable in this case?

Basic rule of counting

Hence for one vowel, there are

```
(i + 1) * (n - i)
```

substrings that include the vowel.

What if we have more than one vowel in our string?

We use the **addition rule**.

Permutations

Permutation

- A permutation is an **arrangement of objects without repetition** where **order is important.**

  Examples

  - Ways to arrange 3 letters: `ABC, ACB, BCA, BAC, CAB, CBA`
  - Ways to arrange x bricks where consecutive bricks have different colors

Permutation

- In general, if we have n objects and want to arrange, then we will have a total of $n!$. If $n!$ is positive integer, then

$$n! = n * (n - 1) * (n - 2) * \ldots * (1)$$

$$n! = n * (n - 1)\,!$$

$$1! = 1$$

Permutation

- If we need to arrange `r` objects (`r ≤ n`) among `n` distinct objects, we will have a total of `P(n, r)` different ways.

```
P(n, r)  = n! / (n - r)!
```

Permutation

In how many ways can we arrange the letters in PEPPER?

Let’s reverse the problem

- How many ways can we arrange $P_1E_1P_2P_3E_2R_1$ (with indices)? $6!$ ways.
- Let’s assume there are x ways to arrange PEPPER (without the indices).
- For a single arrangement, in how many ways can we index the repeated letters?

Answer

`3! * 2! * 1!` ways

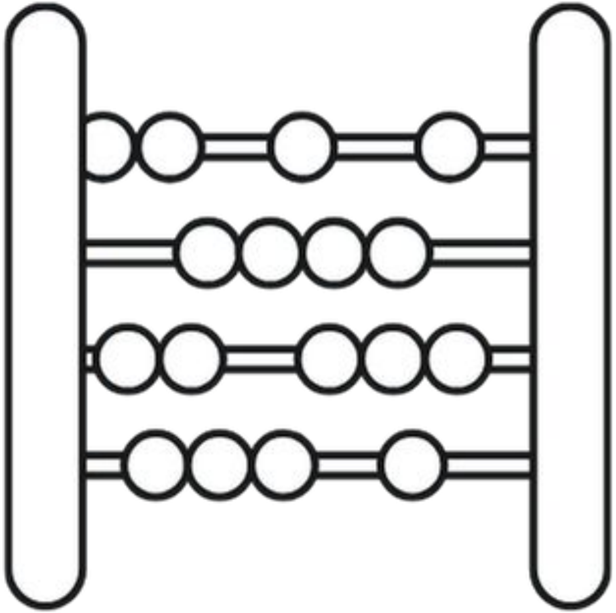
Therefore: `6! =  x * 3! * 2!`

`=> x = 6! / (3! * 2!)`

Permutation

If we need to arrange n objects where `n1, n2, n3 .. nr` are alike. Then we will have

$$\frac{n!}{n_1! \times n_2! \times \cdots \times n_r!}$$



Combinations

Combinations

Combination is **a way of selecting items** from a collection, such that **(unlike permutations)** **the order of selection does not matter.**

Combinations

- Determine the number of different groups of 3 objects from 5 items `A, B, C, D` and `E`.

Combinations

- Every group of 3 will be counted 3! = 6 times. Example: ABC
  - ABC
  - ACB
  - BAC
  - BCA
  - CAB
  - CBA

Combinations

- Initially, there are 5 ways to select the first item.
- Then, there are 4 ways to select the next item.
- Finally, there are 3 ways to select the last item.
- There are thus 5*4*3 ways of selecting the group(if order matters).

Combinations

- The total number of groups that can be formed is:

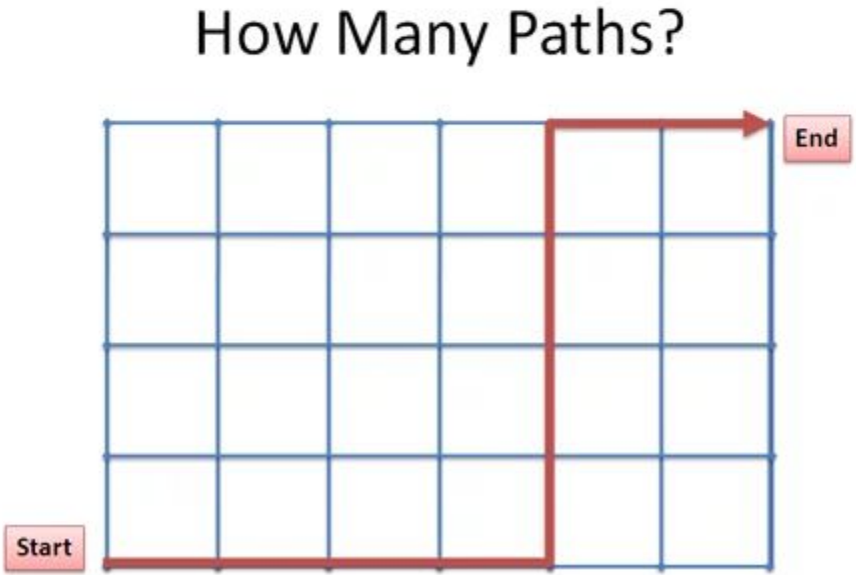
$$\frac{5 \cdot 4 \cdot 3}{3 \cdot 2 \cdot 1} = 10$$

Combinations - Formula

Selecting r objects $(r \leq n)$ among n objects can be done by

$$^{n}C_{r} = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Combinations - Lattice Paths



Suppose you're on a **4 × 6** grid, and want to go from the bottom left to the top right.

How many different paths can you take? We can only move **right** or **up**.

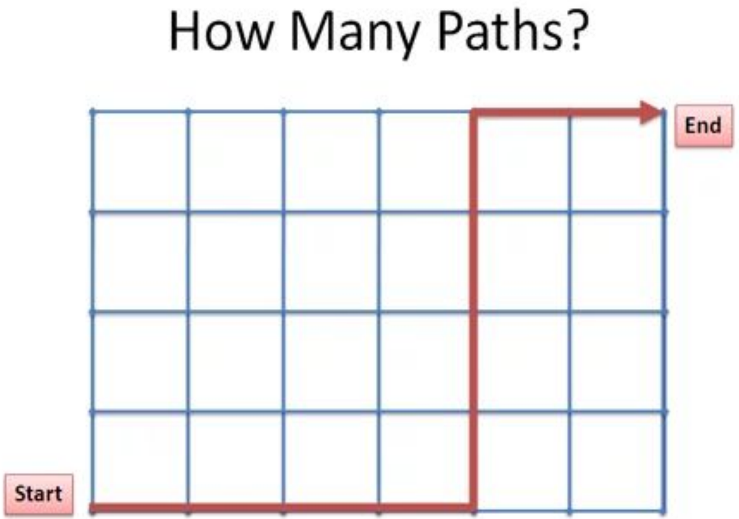
Combinations - Lattice Paths

- Let’s write down a path using “U” (for up) and “R” for (right).
- Path: **R R R R R R U U U U**
- That is, go all the way right (6 R’S) then all the way up (4 U’s)

Combinations - Lattice Paths

- For the diagram the path is

R R R R U U U U R R



Combinations - Lattice Paths

- The question now becomes “In how many ways can we rearrange 4 U’s and 6 R’s.

Combinations - Approach

Imagine we start with **10** R's

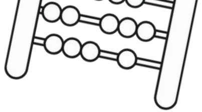
Path: **R R R R R R R R R R**

Clearly, we need to change **4** of those **R**'s into **U**'s.

In how many ways can

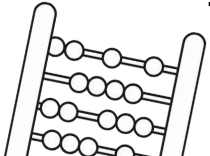
we pick the 4R's

to change them to U's?



Combinations - Lattice Paths

- We have **10** choices for the first **R** to convert
- Then **9** for the second
- Then **8** for the third
- Finally **7** for the last



There are **10 * 9 * 8 * 7 = 5040** possibilities

But Wait!

- We need to remove the redundancies:
- Converting **#1, #2, #3** and **#4** is the same as converting **#4, #3, #2, and #1** or **#3 ,#1, #2 and #4** and soon.
- We have **4!** ways to rearrange the **U**'s positions we picked, so finally we get:
- 5040 / 24 = 210 ways = C(10, 4) = C(10, 6)

Probability

Probability

- **Sample space**:
    - The set of **all possible outcomes** of an experiment.
    - Denoted by `S`.
- Example: If the experiment consists of flipping two coins then

```
S = {(H, H), (H, T), (T, H), (T, T)}
```

Probability

- **Event Space**:
  - The set of **desired** outcomes of the experiment.
  - Denoted by `E`.
- Example: if `E` is the event that a head appears on the first coin, then `E = {(H, H), (H, T)}`

Probability

The probability of an event E is the number of outcomes favourable to E divided by the total number of outcomes.

$$\text{Probability of an event happening} = \frac{\text{Number of ways an event can occur}}{\text{Total number of possible outcomes}}$$

Rules of Probability

- Rule 1: The probability `P(A)` of any event A satisfies `0 <= P(A) <= 1`
- Rule 2: If `S` is the sample space in a probability model, then `P(s) = 1`
- Rule 3: If `A` and `B` are disjoint. `P(A or B) = P(A) + P(B)`
- Rule 4: For any event `A`, `P(A does not occur) = 1 - P(A)`

Probability - Exercise

- Suppose you throw two dice. We are interested in the sum of the upper face of the dice. Let E be the event that the sum of the dice is odd. Find $P(E)$

Conditional Probability

- The conditional probability of an event A is the probability that the event will occur given the knowledge that an event B has already occurred.

Probability of event A and event B

$$P(A|B) = P(A \text{ and } B) / P(B)$$

Probability of event A given B has occurred

Probability of event B

Conditional Probability

- The probability that both A and B have occurred together is

$$P(A \text{ and } B) = P(A|B)\,P(B)$$

Extended Euclidean Algorithm

Extended Euclidean Algorithm

- The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm.
- It finds the greatest common divisor (gcd) of two integers and also finds coefficients (often denoted as x and y) such that

$$ax + by = \gcd(a, b)$$

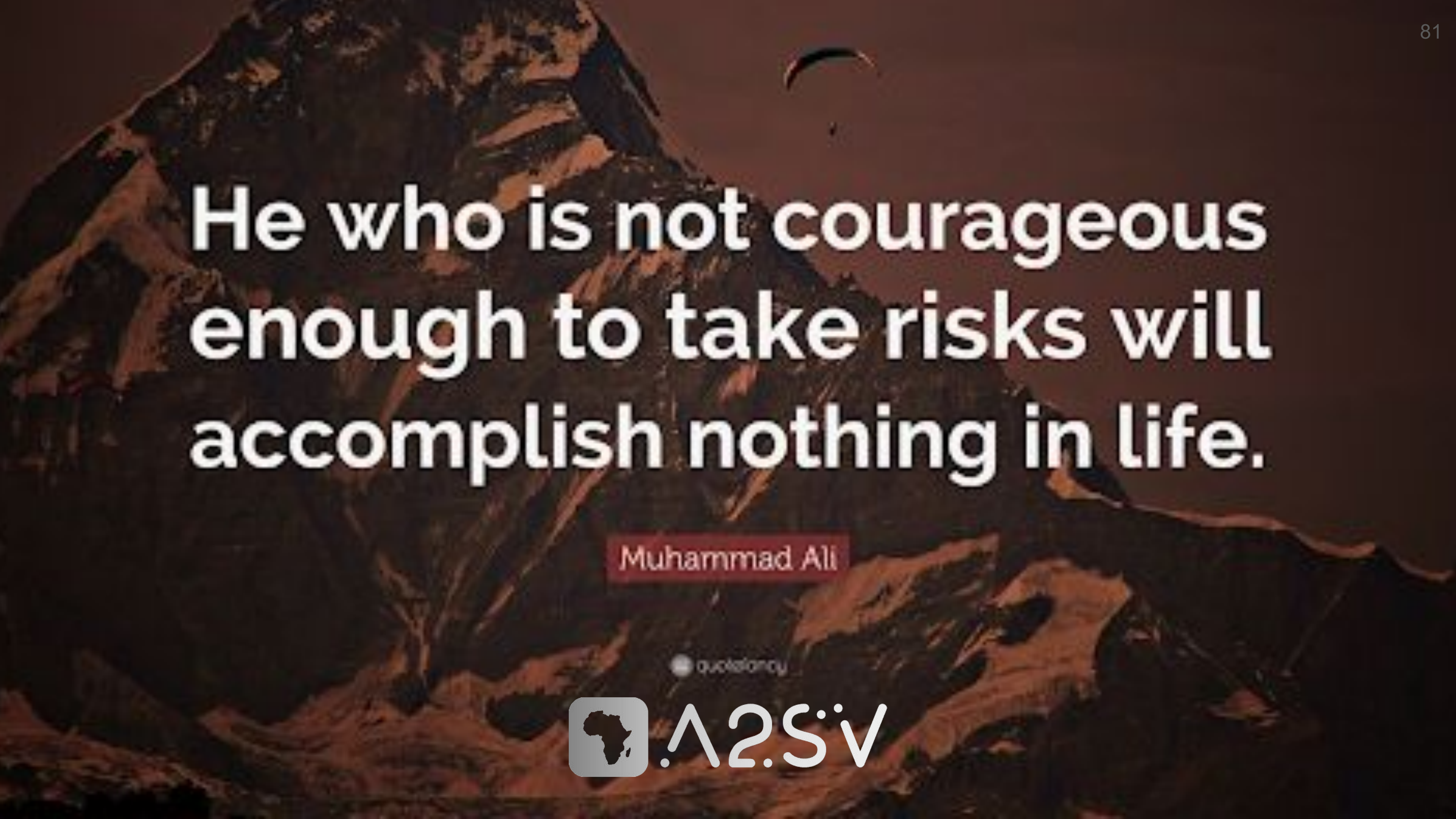
- It is also true that gcd(a, b) is the smallest positive integer for any values of x and y.

Practice Problems

- Pow(x, n)
- Count Good Numbers
- Unique Paths
- Selection of Personnel
- Make it Alternating
- Vowels of All Substrings
- Make Sum Divisible by P
- Password
- Dictionary
- Beautiful Numbers
- Number of Ways to Reach a Position After Exactly k Steps

References

- CP-Algorithms
- Numerics Lecture - A2SV Gen 3 Camp 22
- CP-Algorithms


He who is not courageous enough to take risks will accomplish nothing in life.
Muhammad Ali