<u>**CH4**</u>

Definition of division : If a and b are integers with $a \neq 0$, we say that a divides b if there is an integer c such that $b = ac$ (or equivalently, if b/a is an integer). When a divides b we say that a is a factor or divisor of b, and that b is a multiple of a. The notation $a \mid b$ denotes that a divides b. We write $a \nmid b$ when a does not divide b. Everytime you divide a positive integer there is both a quotient and a remainder. This is defined by the following theorem: Let a be an integer and d a positive integer. Then there are unique integers q and r, with $0 \leq r < d$, such that $a = dq + r$. The quotient and remainder can be expressed by = a div d, r = a mod d. Where d is the divisor, a is the dividend, q is the quotient and r is the remainder. The remainder can also be thought of as the solution to a mod m. The following defines congruency between two expressions: If a and b are integers and m is a positive integer, then a is congruent to b modulo m if m divides $a - b$. We use the notation $a \equiv b \pmod{m}$ to indicate that a is congruent to b modulo m. We say that $a \equiv b \pmod{m}$ is a congruence and that m is its modulus (plural moduli). If a and b are not congruent modulo m, we write $a \not\equiv b \pmod{m}$.

Usually we use decimal notation to express numbers which is a base ten system. It can otherwise be represented in the following way if the positive integer k is equal to 10: Let b be an integer greater than 1. Then if n is a positive integer, it can be expressed uniquely in the form, n = akbk + ak−1bk−1 + ⋯ + a1b + a0, where k is a nonnegative integer, a0, a1, … , ak are nonnegative integers less than b, and ak ≠ 0. Binary expression is when the base is then changed to 2, octal is 8, and hexadecimal is base 16. You can utilize the expansions of integers, particularly binary, to construct several very important algorithms in computer arithmetic like addition and multiplication. In cryptography it is important to be able to find the remainder of an integer raised to an exponent divided by its divisor, because it can be very costly in memory to do this we can instead use the binary expansion of the number to solve with fewer resources.

Every integer greater than 1 that is divisible by itself and 1 is called prime. Numbers greater than one that do not match this definition are called composite. Every integer greater than one can be expressed as a product of primes, this is called its prime factorization. One theorem for identifying large primes is that if n is a composite integer, then n has a prime divisor less than or equal to $\sqrt{n}$. There are an infinite number of primes. One method of finding primes is the sieve of Eratosthenes algorithm. It takes a set of numbers and excludes all composite integers by discarding integers divisible by the sequential integers in the set of positive integers except for the current divisor. We know that prime numbers are infinite but how many prime numbers exist below the integer x? The prime number theorem states this: The ratio of $\pi(x)$, the number of primes not exceeding x, and $x/\ln x$ approaches 1 as x grows without bound. (Here ln x is the natural logarithm of x.) Deriving primes is an important part of cryptography and although many methods, theories, algorithms, and conjectures have been explored and successfully find primes the problem of finding a closed form solution where f(n) is prime for all n has yet to be discovered.

A congruence such as $ax \equiv b \pmod{m}$ is a linear congruence when m is a positive integer, a and b are integers, and x is a variable. It is guaranteed s that an inverse of a modulo m exists whenever a and m are relatively prime by the following theorem: If a and m are relatively prime integers and $m > 1$, then an inverse of a modulo m exists. Furthermore, this inverse is unique modulo m. (That is, there is a unique positive integer $\bar{a}$ less than m that is an inverse of a modulo m and every other inverse of a modulo m is congruent to $\bar{a}$ modulo m.) Systems of linear congruences arise in many contexts one good example is the world puzzle of an ancient Chinese Mathematician that is solved by the Chinese Remainder Theorem: Let m1, m2, … , mn be pairwise relatively prime positive integers greater than one and a1, a2, … , an arbitrary integers. Then the system, x ≡ a1 (mod m1), x ≡ a2 (mod m2), ⋯ x ≡ an (mod mn) has a unique solution modulo m = m1m2 ⋯ mn. (That is, there is a solution x with $0 \leq x < m$, and all other solutions are congruent modulo m to this solution.) Systems of linear congruence can also be used to perform arithmetic with large integers.

One of the practical applications of congruence is through the use of Hashing Functions. A hasing function uses a key to quickly recall or designate the storage location of some information. Its is represented by the function h(k) = k mod m where m is the number of available storage locations. If the location is already taken it under goes a linear probing function to further differentiate the storage location. You can also use congruence to generate pseudorandom numbers with the equation, $x_{n+1} = (ax_n + c) \bmod m$. The number of numbers generated is limited by the number of times the algorithm can execute before returning the original number. One of the most important uses is cryptography. One of the earliest versions of cryptography was simply shifting the index of the letter in the alphabet forward by three. You could decode this message with the knowledge that f(p) = (p + 3) mod 26. The process of recovering plaintext from ciphertext without knowledge of both the encryption method and the key is known as cryptanalysis. To avoid this you must exchange the private key securely if the key is recovered knowing how to encrypt will also reveal how to decrypt except for in public key systems. One of the most well known public encryption systems is the RSA system. This system was developed in secret by the government and encrypts and decrypts the text through an algorithm that converts the message into a representation in integers.

<u>CH. 5</u>

Mathematical induction is used to prove statements that assert that P(n) is Assessment true for all positive integers n, where P(n) is a propositional function. Its made up of two parts the basic step that shows P(1) is true and the inductive step that showsif P (k) is true then all P(k+1) is true. The proof technique is stated as: $(P(1) \wedge \forall k(P(k) \to P(k + 1))) \to \forall nP(n)$. We know that the induction proof is valid because of the ordering property that states every nonempty subset of the set of positive integers has a least element. It is possible to solve problems outside the basic format of mathematical induction as long as the correct basis step is chosen, for example, to show that P(n) is true for n = b, b + 1, b + 2, … , where b is an integer other than 1, we show that P(b) is true in the basis step. In the inductive step, we show that the conditional statement $P(k) \to P(k + 1)$ is true for k = b, b + 1, b + 2, … . Proofs by induction can only be used to prove true conjectures and cannot find new theorems. It is easy to do a false proof by induction if you neglect to properly do both the basis step and the inductive step.

Strong induction requires the same basis step but in the inductive it shows that if the inductive hypothesis P(k) is true, then P(k + 1) is also true. In a proof by strong induction, the inductive step shows that if P(j) is true for all positive integers j not exceeding k, then P(k + 1) is true. That is, for the inductive hypothesis we assume that P(j) is true for j = 1, 2, … , k. Both forms of induction follow the well ordering property. Strong induction is sometimes called the second principle of mathematical induction or complete induction. When deciding to use strong vs normal induction you should ask yourself when it is straightforward to prove that $P(k) \to P(k + 1)$ is true for all positive integers k.

Recursion is used when you cant define something explicitly but can instead define it in terms of itself. To make a recursive solution you specify some initial elements in a basis step (base case) and provide a rule for creating new portions in each iteration. Structural induction is used to prove recursive results. Recursive functions must be well defined meaning that for every positive integer the value of the function at this integer is determined in an unambiguous way. Just like functions can be defined recursively so can sets, they also have the basis and recursive steps but in sets the basis step, an initial collection of elements is specified. In the recursive step, rules for forming new elements in the set from those already known to be in the set are provided. They can also have an exclusion rule that says there are no additional elements apart from those defined in the basis set and elements created by the recursive step.

We can use these ideas of recursion when creating algorithms. An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input. One example is the algorithm to compute factorial ($n! = n \cdot (n - 1)!$). Many iterative solutions can be converted into a recursive algorithm such as linear search and binary search. Mathematical induction and strong induction are used for proofs when it comes to recursive algorithms. Using the base case and successfully applying the recursive definition to find instances of the function at higher values, this is called iterative. Solving iteratively often takes less computational time and effort. A good example of a recursive algorithm is merge sort which can sort values in O(n log n) time.

A program is “correct” if it gives accurate output for every possible input in the domain. To prove this you need two parts, the first showing the program show the right answer when the program ends which gives partial correctness, then the second part shows that the program will in fact always stop. Two propositions are used to prove the correctness of a program, the initial assertion and final assertion. Initial gives properties input has and final gives properties of output. Breaking a program into sub programs and proving each sub is correct proves the whole is correct through the composition rule assuming all sub programs terminate. Conditional statements are logical statements that to be proved to be correct must be true when the initial assertion is true and the final assertion is true or it must be false when the initial is true and the final is true. To develop rules of inference for while loops note that statements continue to execute until the condition is false so the assertion remains true each time the statements interate. This assertion is called loop invariant.