Subject Name: Program Logic Design in 'C'

Semester III **UNIT V**

Computational Model

In computability theory and computational complexity theory, a **model of computation** is the definition of the set of allowable operations used in computation and their respective costs. It is used for measuring the complexity of an algorithm in execution time and or memory space: by assuming a certain model of computation, it is possible to analyze the computational resources required or to discuss the limitations of algorithms or computers.

Some examples of models include Turing machines, finite state machines, recursive functions, lambda calculus, combinatory logic, and abstract rewriting systems.

In the field of runtime analysis of algorithms, it is common to specify a computational model in terms of *primitive operations* allowed which have unit cost, or simply **unit-cost operations**. A commonly used example is the random access machine, which has unit cost for read and write access to all of its memory cells. In this respect, it differs from the above mentioned Turing machine model.

Mathematical induction is a method of mathematical proof typically used to establish a given statement for all natural numbers. It is done in two steps. The first step, known as the **base case**, is to prove the given statement for the first natural number. The second step, known as the **inductive step**, is to prove that the given statement for any one natural number implies the given statement for the next natural number. From these two steps, mathematical induction is the rule from which we infer that the given statement is established for all natural numbers.

The method can be extended to prove statements about more general well-founded structures, such as trees; this generalization, known as structural induction, is used in mathematical logic and computer science. Mathematical induction in this extended sense is closely related to recursion.

Although its namesake may suggest otherwise, mathematical induction should not be misconstrued as a form of inductive reasoning (also see Problem of induction). Mathematical induction is an inference rule used in proofs. In mathematics, proofs are examples of deductive reasoning and inductive reasoning is excluded from proofs.[1]

Description

The simplest and most common form of mathematical induction infers that a statement involving a natural number *n* holds for all values of *n*. The proof consists of two steps:

1. The **basis** (**base case**): prove that the statement holds for the first natural number *n*. Usually, *n* = 0 or *n* = 1.
2. The **inductive step**: prove that, if the statement holds for some natural number *n*, then the statement holds for *n* + 1.

The hypothesis in the inductive step that the statement holds for some *n* is called the **induction hypothesis** (or **inductive hypothesis**). To perform the inductive step, one assumes the induction hypothesis and then uses this assumption to prove the statement for *n* + 1.

Whether *n* = 0 or *n* = 1 depends on the definition of the natural numbers. If 0 is considered a natural number, as is common in the fields of combinatorics and mathematical logic, the base case is given by *n* = 0. If, on the other hand, 1 is taken as the first natural number, then the base case is given by *n* = 1.

Example

Mathematical induction can be used to prove that the following statement, which we will call *P*(*n*), holds for all natural numbers *n*.

P(*n*) gives a formula for the sum of the natural numbers less than or equal to number *n*. The proof that *P*(*n*) is true for each natural number *n* proceeds as follows.

Basis: Show that the statement holds for *n* = 0.
P(0) amounts to the statement:

In the left-hand side of the equation, the only term is 0, and so the left-hand side is simply equal to 0.
In the right-hand side of the equation, 0·(0 + 1)/2 = 0.
The two sides are equal, so the statement is true for *n* = 0. Thus it has been shown that *P*(0) holds.

Inductive step: Show that *if P*(*k*) holds, then also *P*(*k* + 1) holds. This can be done as follows.

Assume *P*(*k*) holds (for some unspecified value of *k*). It must then be shown that *P*(*k* + 1) holds, that is:

$$(0 + 1 + 2 + \cdots + k) + (k + 1) = \frac{(k + 1)((k + 1) + 1)}{2}.$$

Using the induction hypothesis that *P*(*k*) holds, the left-hand side can be rewritten to:

$$\frac{k(k + 1)}{2} + (k + 1)\,.$$

Algebraically:

$$\begin{aligned}\frac{k(k+1)}{2}+(k+1) &= \frac{k(k+1)+2(k+1)}{2} \\ &= \frac{k^2+k+2k+2}{2} \\ &= \frac{(k+1)(k+2)}{2} \\ &= \frac{(k+1)((k+1)+1)}{2}\end{aligned}$$

thereby showing that indeed $P(k+1)$ holds.

Since both the basis and the inductive step have been performed, by mathematical induction, the statement $P(n)$ holds for all natural n. Q.E.D.

Axiom of induction

Mathematical induction as an inference rule can be formalized as a second-order axiom. The *axiom of induction* is, in logical symbols,

$$\forall P[[P(0) \wedge \forall k \in \mathbb{N}(P(k) \Rightarrow P(k+1))] \Rightarrow \forall n \in \mathbb{N}[P(n)]]$$

where P is any predicate and k and n are both natural numbers.

In words, the basis $P(0)$ and the inductive step (namely, that the inductive hypothesis $P(k)$ implies $P(k+1)$) together imply that $P(n)$ for any natural number n. The axiom of induction asserts that the validity of inferring that $P(n)$ holds for any natural number n from the basis and the inductive step.

Note that the first quantifier in the axiom ranges over *predicates* rather than over individual numbers. This is a second-order quantifier, which means that this axiom is stated in second-order logic. Axiomatizing arithmetic induction in first-order logic requires an axiom schema containing a separate axiom for each possible predicate. The article Peano axioms contains further discussion of this issue.

Heuristic justification

As an inference rule, mathematical induction can be justified as follows. Having proven the base case and the inductive step, then any value can be obtained by performing the inductive step repeatedly. It may be helpful to think of the domino effect. Consider a half line of dominoes each standing on end, and extending infinitely to the right. Suppose that:

1. The first domino falls right.
2. If a (fixed but arbitrary) domino falls right, then its next neighbor also falls right.

With these assumptions one can conclude (using mathematical induction) that all of the dominoes will fall right.

Mathematical induction, as formalized in the second-order axiom above, works because k is used to represent an *arbitrary* natural number. Then, using the inductive hypothesis, i.e. that $P(k)$ is true, show $P(k + 1)$ is also true. This allows us to "carry" the fact that $P(0)$ is true to the fact that $P(1)$ is also true, and carry $P(1)$ to $P(2)$, etc., thus proving $P(n)$ holds for every natural number n.

Variants

This section includes a list of references, related reading or external links, but **the sources of this section remain unclear because it lacks inline citations**. Please improve this article by introducing more precise citations. *(July 2013)*

In practice, proofs by induction are often structured differently, depending on the exact nature of the property to be proved.

Starting at some other number

If we want to prove a statement not for all natural numbers but only for all numbers greater than or equal to a certain number b then:

1. Showing that the statement holds when $n = b$.
2. Showing that if the statement holds for $n = m \geq b$ then the same statement also holds for $n = m + 1$.

This can be used, for example, to show that $n^2 \geq 3n$ for $n \geq 3$. A more substantial example is a proof that

$$\frac{n^n}{3^n} < n! < \frac{n^n}{2^n} \text{ for } n \geq 6.$$

In this way we can prove that $P(n)$ holds for all $n \geq 1$, or even $n \geq -5$. This form of mathematical induction is actually a special case of the previous form because if the statement that we intend to prove is $P(n)$ then proving it with these two rules is equivalent with proving $P(n + b)$ for all natural numbers n with the first two steps.

Building on $n = 2$

In mathematics, many standard functions, including operations such as "+" and relations such as "=", are binary, meaning that they take two arguments. Often these functions possess properties that implicitly extend them to more than two arguments. For example, once addition $a + b$ is defined and is known to satisfy the associativity property $(a + b) + c = a + (b + c)$, then the ternary addition $a + b + c$ makes sense, either as $(a + b) + c$ or as $a + (b + c)$. Similarly, many axioms and theorems in mathematics are stated only for the binary versions of mathematical operations and relations, and implicitly extend to higher-arity versions.

Suppose that we wish to prove a statement about an n-ary operation implicitly defined from a binary operation, using mathematical induction on n. Then it should come as no surprise that the $n = 2$ case carries special weight. Here are some examples.

Example: product rule for the derivative

In this example, the binary operation in question is multiplication (of functions). The usual product rule for the derivative taught in calculus states:

$$(fg)' = f'g + g'f.$$

or in logarithmic derivative form

$$(fg)'/(fg) = f'/f + g'/g.$$

This can be generalized to a product of *n* functions. One has

$$\begin{aligned}&(f_1 f_2 f_3 \cdots f_n)' \\ &= (f_1' f_2 f_3 \cdots f_n) + (f_1 f_2' f_3 \cdots f_n) + (f_1 f_2 f_3' \cdots f_n) + \cdots + (f_1 f_2 \cdots f_{n-1} f_n').\end{aligned}$$

or in logarithmic derivative form

$$\begin{aligned}&(f_1 f_2 f_3 \cdots f_n)'/(f_1 f_2 f_3 \cdots f_n) \\ &= (f_1'/f_1) + (f_2'/f_2) + (f_3'/f_3) + \cdots + (f_n'/f_n).\end{aligned}$$

In each of the *n* terms of the usual form, just one of the factors is a derivative; the others are not.

When this general fact is proved by mathematical induction, the *n* = 0 case is trivial, $(1)' = 0$ (since the empty product is 1, and the empty sum is 0). The *n* = 1 case is also trivial, $f_1' = f_1'$. And for each *n* ≥ 3, the case is easy to prove from the preceding *n* − 1 case. The real difficulty lies in the *n* = 2 case, which is why that is the one stated in the standard product rule.

An alternative way to look at this is to generalize $f(xy) = f(x) + f(y), f(1) = 0$ (a monoid homomorphism) to $f(\prod x_i) = \sum f(x_i)$.

Example: Pólya's proof that there is no "horse of a different color"

Main article: All horses are the same color

In this example, the binary relation in question is an equivalence relation applied to horses, such that two horses are equivalent if they are the same color. The argument is essentially identical to the one above, but the crucial *n* = 1 case fails, causing the entire argument to be invalid.

In the middle of the 20th century, a commonplace colloquial locution to express the idea that something is unexpectedly different from the usual was "*That's* a horse of a different color!". George Pólya posed the following exercise: Find the error in the following argument, which purports to prove by mathematical induction that all horses are of the same color:

- Basis: If there is only *one* horse, there is only one color.
- Induction step: Assume as induction hypothesis that within any set of n horses, there is only one color. Now look at any set of $n + 1$ horses. Number them: 1, 2, 3, ..., n, $n + 1$. Consider the sets {1, 2, 3, ..., n} and {2, 3, 4, ..., $n + 1$}. Each is a set of only n horses, therefore within each there is only one color. But the two sets overlap, so there must be only one color among all $n + 1$ horses.

The basis case n=1 is trivial (as any horse is the same color as itself), and the inductive step is correct in all cases $n > 1$. However, the logic of the inductive step is incorrect for $n = 1$, because the statement that "the two sets overlap" is false (there are only n+1=2 horses prior to either removal, and after removal the sets of one horse each do not overlap). Indeed, going from the $n = 1$ case to the $n = 2$ case is clearly the crux of the matter; if one could prove the $n = 2$ case directly without having to infer it from the n=1 case, then all higher cases would follow from the inductive hypothesis.

Induction on more than one counter

It is sometimes desirable to prove a statement involving two natural numbers, n and m, by iterating the induction process. That is, one performs a basis step and an inductive step for n, and in each of those performs a basis step and an inductive step for m. See, for example, the proof of commutativity accompanying *addition of natural numbers*. More complicated arguments involving three or more counters are also possible.

Infinite descent

Main article: Infinite descent

The method of infinite descent was one of Pierre de Fermat's favorites. This method of proof can assume several slightly different forms. For example, it might begin by showing that if a statement is true for a natural number n it must also be true for some smaller natural number m ($m < n$). Using mathematical induction (implicitly) with the inductive hypothesis being that the statement is false for all natural numbers less than or equal to m, we can conclude that the statement cannot be true for any natural number n.

Although this particular form of infinite-descent proof is clearly a mathematical induction, whether one holds all proofs "by infinite descent" to be mathematical inductions depends on how one defines the term "proof by infinite descent." One might, for example, use the term to apply to proofs in which the well-ordering of the natural numbers is assumed, but not the principle of induction. Such, for example, is the usual proof that 2 has no rational square root (see Infinite descent).

Complete induction

Another variant, called **complete induction** (or **strong induction** or **course of values induction**), says that in the second step we may assume not only that the statement holds for $n = m$ but also that it is true for **all** n less than or equal to m.

Complete induction is most useful when several instances of the inductive hypothesis are required for each inductive step. For example, complete induction can be used to show that

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi}$$

where F_n is the n^{th} Fibonacci number, $\phi = (1 + \sqrt{5})/2$ (the golden ratio) and $\psi = (1 - \sqrt{5})/2$ are the roots of the polynomial $x^2 - x - 1$. By using the fact that $F_{n+2} = F_{n+1} + F_n$ for each $n \in \mathbf{N}$, the identity above can be verified by direct calculation for F_{n+2} if we assume that it already holds for both F_{n+1} and F_n. To complete the proof, the identity must be verified in the two base cases $n = 0$ and $n = 1$.

Another proof by complete induction uses the hypothesis that the statement holds for *all* smaller n more thoroughly. Consider the statement that "every natural number greater than 1 is a product of prime numbers", and assume that for a given $m > 1$ it holds for all smaller $n > 1$. If m is prime then it is certainly a product of primes, and if not, then by definition it is a product: $m = n_1 n_2$, where neither of the factors is equal to 1; hence neither is equal to m, and so both are smaller than m. The induction hypothesis now applies to n_1 and n_2, so each one is a product of primes. Then m is a product of products of primes; i.e. a product of primes.

This generalization, complete induction, is equivalent to the ordinary mathematical induction described above. Suppose P(n) is the statement that we intend to prove by complete induction. Let Q(n) mean P(m) holds for all m such that $0 \le m \le n$. Then Q(n) is true for all n if and only if P(n) is true for all n, and a proof of P(n) by complete induction is just the same thing as a proof of Q(n) by (ordinary) induction.

Transfinite induction

Main article: Transfinite induction

The last two steps can be reformulated as one step:

1. Showing that if the statement holds for all $n < m$ then the same statement also holds for $n = m$.

This form of mathematical induction is not only valid for statements about natural numbers, but for statements about elements of any well-founded set, that is, a set with an irreflexive relation < that contains no infinite descending chains.

This form of induction, when applied to ordinals (which form a well-ordered and hence well-founded class), is called *transfinite induction*. It is an important proof technique in set theory, topology and other fields.

Proofs by transfinite induction typically distinguish three cases:

1. when m is a minimal element, i.e. there is no element smaller than m
2. when m has a direct predecessor, i.e. the set of elements which are smaller than m has a largest element
3. when m has no direct predecessor, i.e. m is a so-called limit-ordinal

Strictly speaking, it is not necessary in transfinite induction to prove the basis, because it is a vacuous special case of the proposition that if P is true of all $n < m$, then P is true of m. It is vacuously true precisely because there are no values of $n < m$ that could serve as counterexamples.

Equivalence with the well-ordering principle

The principle of mathematical induction is usually stated as an axiom of the natural numbers; see Peano axioms. However, it can be proved from the well-ordering principle. Indeed, suppose the following:

- The set of natural numbers is well-ordered.
- Every natural number is either zero, or n+1 for some natural number n.
- For any natural number n, n+1 is greater than n.

To derive simple induction from these axioms, we must show that if P(n) is some proposition predicated of n, and if:

- P(0) holds and
- whenever P(k) is true then P(k+1) is also true

then P(n) holds for all n.

Proof. Let S be the set of all natural numbers for which P(n) is false. Let us see what happens if we assert that S is nonempty. Well-ordering tells us that S has a least element, say t. Moreover, since P(0) is true, t is not 0. Since every natural number is either zero or some n+1, there is some natural number n such that n+1=t. Now n is less than t, and t is the least element of S. It follows that n is not in S, and so P(n) is true. This means that P(n+1) is true, and so P(t) is true. This is a contradiction, since t was in S. Therefore, S is empty.

It can also be proved that induction, given the other axioms, implies the well-ordering principle.

notion of algorithm

The development of a computer program takes place in several steps.

This is to provide a solution to a problem, the first step is to analyze the problem, ie, identifying the limits of the exercise in the descriptive language that usually comes in the analysis to describe the process by which the problem is formalized. Description language used to write the test result is called an algorithm. The next step is to translate the algorithm on a specific programming language, which is the programming phase.

The programming language is the intermediary between man and machine, allowing you to write in a language close to the machine, but the intelligible human operations that the computer must meet. Therefore, given that the programming language is designed for the computer, you must follow a strict syntax. An algorithm can still lead to several programs.

The program is then converted into machine language at a stage called compilation. compilation phase is performed by the computer itself by another program called a compiler.

The next step is called the linker, which link the program with all external components (usually called libraries).
Characteristics of an algorithm

The algorithm is a way for the programmer to make your approach to the problem to others. Indeed, an algorithm is expressed in a well-defined series of steps to solve the problem of the language area. An algorithm must be:

read: the algorithm should be understandable even for non-computer high-level algorithm must be translated into a programming language, so it should not make use of technical terms related to a program or a particular operating system Specific: each element of the algorithm is not to be confused, and it is important to remove any ambiguity
concise: an algorithm should not exceed one page. If this is the case, we decompose the

problem into sub-problems
structured: an algorithm must be composed of different parts easily identifiable

A recursive method is a method that calls itself either directly or indirectly

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations.

Iteration Vs. Recursion

- If a recursive method is called with a base case, the method returns a result. If a method is called with a more complex problem, the method divides the problem into two or more conceptual pieces: a piece that the method knows how to do and a slightly smaller version of the original problem. Because this new problem looks like the original problem, the method launches a recursive call to work on the smaller problem.
- For recursion to terminate, each time the recursion method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. When the method recognizes the base case, the result is returned to the previous method call and a sequence of returns ensures all the way up the line until the original call of the method eventually returns the final result.
- Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated method calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.
- Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls. This can be expensive in both processor time and memory space.

Difference between Recursion and Iteration

RECURSION	**ITERATIONS**
Recursive function – is a function that is partially defined by itself	Iterative Instructions –are loop based repetitions of a process
Recursion Uses selection structure	Iteration uses repetition structure
Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on some condition.(base case)	An infinite loop occurs with iteration if the loop-condition test never becomes false
Recursion terminates when a base case is recognized	Iteration terminates when the loop-condition fails
Recursion is usually slower then iteration due to overhead of maintaining stack	Iteration does not use stack so it's faster than recursion
Recursion uses more memory than iteration	Iteration consume less memory
Infinite recursion can crash the system	infinite looping uses CPU cycles repeatedly
Recursion makes code smaller	Iteration makes code longer