

UNIT I

ADVANCED DATA STRUCTURES AND ALGORITHMS

UNIT I

ITERATIVE AND RECURSIVE ALGORITHMS

- Iterative Algorithms: Measures of Progress and Loop Invariants-Paradigm Shift: Sequence of Actions versus Sequence of Assertions- Steps to Develop an Iterative Algorithm-Different Types of Iterative Algorithms--Typical Errors-Recursion-Forward versus Backward- Towers of Hanoi Checklist for Recursive Algorithms-The Stack Frame-Proving Correctness with Strong Induction Examples of Recursive Algorithms-Sorting and Selecting Algorithms-Operations on Integers Ackermann's Function- Recursion on Trees-Tree Traversals- Examples- Generalizing the Problem - Heap Sort and Priority Queues-Representing Expressions.

ITERATIVE ALGORITHMS

- **A Computational Problem:** A specification of a computational problem uses *preconditions* and *postconditions* to describe for each legal input instance that the computation might receive, what the required output or actions are.
- It may be an optimization problem which requires a solution to be outputted that is “optimal” from among a huge set of possible solutions for the given input instance.
- **Example: The sorting problem** is defined as follows:
- *Preconditions:* The input is a list of n values, including possible repetitions.
- *Post conditions:* The output is a list consisting of the same n values in non decreasing order.

ITERATIVE ALGORITHMS

- **An Algorithm:** An *algorithm is a step-by-step procedure which, starting with an input instance*, produces a suitable output.
- **Correctness:** An algorithm for the problem is *correct if for every legal input instance*, the required output is produced
- **Running Time:** It is not enough for a computation to eventually get the correct answer. It must also do so using a reasonable amount of time and memory space. The *running time of an algorithm is a function from the size n of the input stance given to a bound on the number of operations the computation must do*.
- *The algorithm is said to be feasible if this function is a polynomial lime $\text{Time}(n) = \Theta(n^2)$, and is said to be infeasible if the function is an exponential lime $\text{Time}(n) = \Theta(2^n)$*

ITERATIVE ALGORITHMS

- An iterative algorithm (Part One) takes one step at a time, ensuring that each step makes *progress while maintaining the loop invariant*.
- A recursive algorithm breaks its instance into smaller instances, which it gets a *friend to solve*, and then combines their solutions into one of its own.
- *Optimization problems form an important class of computational problems.*
- *Dynamic programming solves a sequence of larger and larger instances, reusing the previously saved solutions for the smaller instances, until a solution is obtained for the given instance.*

MEASURES OF PROGRESS AND LOOP INVARIANT

- A *measure of progress* is how far you are either from your starting location (point) or from your destination.
- Iterative algorithms example:
- $\text{Max}(a, b, c)$
- ***PreCond: Input has 3 numbers.***
- $m = a$
- *assert: m is max in {a}.*
- $\text{if}(b > m)$
- $m = b$
- end if
- *assert: m is max in {a,b}.*
- $\text{if}(c > m)$
- $m = c$
- end if
- *assert: m is max in {a,b,c}.*
- $\text{return}(m)$
- ***PostCond: return max in {a,b,c}***
- end algorithm

THE STEPS TO DEVELOP AN ITERATIVE ALGORITHM

- ❑ **Loop Invariant:** A *loop invariant expresses important relationships among the* variables that must be true at the start of every iteration and when the loop terminates.
 - If it is true, then the computation is still on the road. If it is false, then the algorithm has failed.
- ❑ **The Code Structure:** The basic structure of the code is as follows.
 - begin routine
 - *pre-cond*
 - *Code_{pre-loop} % Establish loop invariant*
 - loop
 - *<loop-invariant >*
 - exit when *<exit-cond >*
 - *Code_{loop} % Make progress while maintaining the loop invariant*
 - end loop
 - *Code_{post-loop} % Clean up loose ends*
 - *post-cond*
 - end routine

THE STEPS TO DEVELOP AN ITERATIVE ALGORITHM

- ❑ **Proof of Correctness:** The algorithm will work on all specified inputs and give the correct answer.
- ❑ **Running Time:** The algorithm will complete in a reasonable amount of time.

ITERATIVE ALGORITHMS










<p>Define Problem</p> 	<p>Define Loop Invariants</p> 	<p>Define Measure of Progress</p> 
<p>Define Step</p> 	<p>Define Exit Condition</p> 	<p>Maintain Loop Inv</p> 
<p>Make Progress</p> 	<p>Initial Conditions</p> 	<p>Ending</p> 

Figure 1.1: The requirements of an iterative algorithm.

THE STEPS TO DEVELOP AN ITERATIVE ALGORITHM

- **1) Specifications:** It describes about the pre- and post conditions
 - —i.e., where are you starting and where is your destination?
- **2) Basic Steps:** What basic steps will head you more or less in the correct direction?
- **3) Measure of Progress:** You must define a measure of progress: where are the mile markers along the road?
i.e How long you have to travel to achieve your destination
- **4) The Loop Invariant:** You must define a loop invariant that will give a picture of the state of your computation when it is at the top of the main loop, in other words, define the road that you will stay on.
- **5) Main Steps:** Write pseudocode for every single step

THE STEPS TO DEVELOP AN ITERATIVE ALGORITHM

- **6) Make Progress:** Each iteration of your main step must make progress according to your measure of progress.
- **7) Maintain Loop Invariant:** Each iteration of your main step must ensure that the loop invariant is true again when the computation gets back to the top of the loop.
- **8) Establishing the Loop Invariant:** Now that you have an idea of where you are going, you have a better idea about how to begin.

THE STEPS TO DEVELOP AN ITERATIVE ALGORITHM

- **9) Exit Condition:** write the condition *exit-cond* that causes the computation to break out of the loop.
- **10) Ending:** How does the exit condition together with the invariant ensure that the problem is solved? When at the end of the road but still on it, how do you produce the required output? You must write the pseudo code *code post-loop* to clean up loose ends and to return the required output.
- **11) Termination and Running Time:** Where to terminate your program and calculate the running time of your algorithm
- **12) Special Cases:** When first attempting to design an algorithm, you should only consider one general type of input instances. Later, you must cycle through the steps again considering other types of instances and special cases. Similarly, test your algorithm by hand on a number of different examples.
- **13) Coding and Implementation Details:** Now you are ready to put all the pieces together and produce pseudocode for the algorithm. It may be necessary at this point to provide extra implementation details.
- **14) Formal Proof:** If the above pieces fit together as required, then your algorithm works.

THE STEPS TO DEVELOP AN ITERATIVE ALGORITHM - DIFFERENT TYPES OF ITERATIVE ALGORITHMS

- **Example - Binary Search**
- **1) Specifications:** An input instance consists of a sorted list $A[1..n]$ of elements and a key to be searched for. Elements may be repeated. If the key is in the list, then the output consists of an index i such that $A[i] = \text{key}$. If the key is not in the list, then the output reports this.
- **2) Basic Steps:** Continue to cut the search space in which the key might be in half.
- **4) The Loop Invariant:** The algorithm maintains a sublist $A[i..j]$ such that if the key is contained in the original list $A[1..n]$, then it is contained in this narrowed sublist. (If the element is repeated, then it might also be outside this sublist.)
- **3) Measure of Progress:** The measure of progress is the number of elements in our sublist, namely $j - i + 1$.

DIFFERENT TYPES OF ITERATIVE ALGORITHMS - EXAMPLE

- **5) Main Steps:** Each iteration compares the key with the element at the center of the sublist. This determines which half of the sublist the key is not in and hence which half to keep. More formally, let mid index the element in the middle of our current sublist $A[i..j]$. If $key \leq A[mid]$, then the sublist is narrowed to $A[i..mid]$. Otherwise, it is narrowed to $A[mid + 1..j]$.
- **6) Make Progress:** The size of the sublist decreases by a factor of two.
- **7) Maintain Loop Invariant:** *loop-invariant & not exit-cond & code loop* \Rightarrow *loop-invariant*. The previous loop invariant gives that the search has been narrowed down to the sublist $A[i..j]$. If $key > A[mid]$, then because the list is sorted, we know that key is not in $A[i..mid]$ and hence these elements can be thrown away, narrowing the search to $A[mid + 1..j]$. Similarly if $key < A[mid]$. If $key = A[mid]$, then we could report that the key has been found. However, the loop invariant is also
- maintained by narrowing the search down to $A[i..mid]$.

DIFFERENT TYPES OF ITERATIVE ALGORITHMS - EXAMPLE

- **8) Establishing the Loop Invariant:** *pre-cond & code_{pre-loop} => loop-invariant* .
- Initially, you obtain the loop invariant by considering the entire list as the sublist. It trivially follows that if the key is in the entire list, then it is also in this sublist.
- **9) Exit Condition:** We exit when the sublist contains one (or zero) elements.
- **10) Ending:** *loop-invariant & exit-cond & code_{post-loop} => post-cond* . By the exit condition, our sublist contains at most one element, and by the loop invariant, if the key is contained in the original list, then the key is contained in this sublist, i.e., must
- be this one element. Hence, the final code tests to see if this one element is the key. If it is, then its index is returned. If it is not, then the algorithm reports that the key is not in the list.

DIFFERENT TYPES OF ITERATIVE ALGORITHMS - EXAMPLE

- **11) Termination and Running Time:** The sizes of the sublists are approximately $n, n/2, n/4, n/8, n/16, \dots, 8, 4, 2, 1$. Hence, only $(\log n)$ splits are needed. Each split takes $O(1)$ time. Hence, the total time is $(\log n)$.
- **12) Special Cases:** A special case to consider is when the key is not contained in the original list $A[1..n]$. Note that the loop invariant carefully takes this case into account. The algorithm will narrow the sublist down to one (or zero) elements. The counter positive of the loop invariant then gives that if the key is not contained in this narrowed sublist, then the key is not contained in the original list $A[1..n]$.
- **13) Coding and Implementation Details:** In addition to testing whether $key \leq A[mid]$, each iteration could test to see if $A[mid]$ is the key. Though finding the key in this way would allow you to stop early, extensive testing shows that this extra comparison slows down the computation.

TYPICAL ERRORS

- ❑ **Be Clear:** The code specifies the current subinterval $A[i.. j]$ with two integers i and j . Clearly document whether the sublist includes the end points i and j or not. It does not matter which, but you must be consistent. Confusion in details like this is the cause of many bugs.
- ❑ **Math Details:** Small math operations like computing the index of the middle element of the subinterval $A(i.. j)$ are prone to bugs. Check for yourself that the answer is $mid = i+j$
- ❑ **Make Progress:** Be sure that each iteration progress is made in every special case. For example, in binary search, when the current sublist has even length, it is reasonable (as done above) to let mid be the element just to the left of center. It is also reasonable to include the middle element in the right half of the sublist. However, together these cause a bug. Given the sublist $A[i.. j] = A[3, 4]$, the middle will be the element indexed with 3, and the right sublist will be still be $A[mid.. j] = A[3, 4]$. If this sublist is kept, no progress will be made, and the Algorithm will loop forever.

TYPICAL ERRORS

- ❑ **Maintain Loop Invariant:** Be sure that the loop invariant is maintained in every special case. For example, in binary search, it is reasonable to test whether $key < A[mid]$ or $key \geq A[mid]$. It is also reasonable for it to cut the sublist $A[i..j]$ into $A[i..mid]$ and $A[mid + 1..j]$. However, together these cause a bug. When key and $A[mid]$ are equal, the test $key < A[mid]$ will fail, causing the algorithm to think the key is bigger and to keep the right half $A[mid + 1..j]$. However, this skips over the key.
- ❑ **Simple Loop:** Code like “ $i = 1; \text{while}(i \leq n) A[i] = 0; i = i + 1; \text{end while}$ ” is surprisingly prone to the error of being off by one. The loop invariant “When at the top of the loop, i indexes the next element to handle” helps a lot.

RECURSION – LOOKING FORWARD VS. BACKWARD

- A function call by itself is called “Recursion”
- **Circular Argument:** Recursion involves designing an algorithm by using it as if it already exists. At first this looks paradoxical. Suppose, for example, the key to the house that you want to get into is in that same house. If you could get in, you could get the key. Then you could open the door, so that you could get in. This is a circular argument. It is not a legal recursive program because the sub instance is not smaller.

LOOKING FORWARD VS. BACKWARD - EXAMPLE

- **One Problem and a Row of Instances:**
- Consider a row of houses. Each house is bigger than the next. Your task is to get into the biggest one. You are locked out of all the houses. The key to each house is locked in the house of the next smaller size. The recursive problem consists in getting into any specified house. Each house in the row is a separate instance of this problem. To get into my house I must get the key from a smaller house



LOOKING FORWARD VS. BACKWARD - EXAMPLE

- **The Algorithm:** The smallest house is small enough that one can use brute force to get in. For example, one could simply lift off the roof. Once in this house, we can get the key to the next house, which is then easily opened. Within this house, we can get the key to the house after that, and so on. Eventually, we are in the largest house as required.

LOOKING FORWARD VS. BACKWARD - EXAMPLE

- **Working Forward vs. Backward:** An iterative algorithm works forward. It knows about house $i - 1$. It uses a loop invariant to show that this house has been opened. It searches this house and learns that the key within it is that for house i . Because of this, it decides that house i would be a good one to go to next.
- A recursion algorithm works backward. It knows about house i . It wants to get it open. It determines that the key for house i is contained in house $i - 1$. Hence, opening house $i - 1$ is a subtask that needs to be accomplished.
- There are two advantages of recursive algorithms over iterative ones. The first is that sometimes it is easier to work backward than forward. The second is that a recursive algorithm is allowed to have more than one subtask to be solved. This forms a tree of houses to open instead of a row of houses.

THE TOWERS OF HANOI

- **Specification:** The puzzle consists of three poles and a stack of N disks of different sizes.
- **Precondition:** All the disks are on the first of the three poles.



Figure 8.1: The towers of Hanoi problem.

- **Postcondition:** The goal is to move the stack over to the last pole. See the first and the last parts of Figure 8.1.

THE TOWERS OF HANOI

- You are only allowed to take one disk from the top of the stack on one pole and place it on the top of the stack on another pole. Another rule is that no disk can be placed on top of a smaller disk.
- **Lost with First Step:** The first step must be to move the smallest disk. But it is by no means clear whether to move it to the middle or to the last pole.

THE TOWERS OF HANOI

- **Divide: Jump into the middle of the computation.** One thing that is clear is that at some point, you must move the biggest disk from the first pole to the last. In order to do this, there can be no other disks on either the first or the last pole. Hence, all the other disks need to be stacked on the middle pole.
- See the second and the third parts of Figure 8.1. This point in the computation splits the problem into two sub problems that must be solved. The first is how to move all the disks except the largest from the first pole to the middle. See the first and second parts of Figure 8.1. The second is how to move these same disks from the middle pole to the last. See the third and fourth parts of Figure 8.1.

THE TOWERS OF HANOI

- **Conquer:** Together these steps solve the entire problem. Starting with all disks on the first pole, somehow move all but the largest to the second pole. Then, in one step, move the largest from the first to the third pole. Finally, somehow move all but the largest from the second to the third pole.
- **More General Specification:** The sub problem of moving all but the largest disk from the first to the middle pole is very similar to original towers of Hanoi problem.
- However, it is an instance of a slightly more general problem, because not all of the disks are moved. To include this as an instance of our problem, we generalize the problem as follows.

THE TOWERS OF HANOI

- **Precondition:** *The input specifies the number n of disks to be moved and the roles of the three poles. These three roles for poles are $pole_{source}$, $pole_{destination}$,*
- *and $pole_{spare}$. The precondition requires that the smallest n disks be currently on $pole_{source}$. It does not care where the larger disks are.*
- **Postcondition:** *The goal is to move these smallest n disks to , $pole_{destination}$. Pole $Pole_{spare}$ is available to be used temporarily. The larger disks are not moved.*

THE TOWERS OF HANOI

- **Code:**
- algorithm *TowersOfHanoi(n, source, destination, spare)*
- **<pre-cond>**: *The n smallest disks are on pole_{source}.*
- **<post-cond>**: *They are moved to pole_{destination}.*
- begin
- if($n \leq 0$)
- Nothing to do
- else
- *TowersOfHanoi(n-1, source, spare, destination)*
- Move the *n*th disk from pole_{source} to pole_{destination}.
- *TowersOfHanoi(n-1, spare, destination, source)*
- end if
- end algorithm
- **Running Time:** Let $T(n)$ be the time to move n disks. Clearly, $T(1) = 1$ and $T(n) = 2 \cdot T(n-1) + 1$. Solving this gives $T(n) = 2^n - 1$.

CHECKLIST FOR RECURSIVE ALGORITHMS

- This section contains a list of things to think about to make sure that you do not make any of the common mistakes
- **o) The Code Structure:** The code does not need to be much more complex than the following.
- **algorithm** $Alg(a, b, c)$
- ***pre-cond*:** Here a is a tuple, b an integer, and c a binary tree.
- ***post-cond*:** Outputs x , y , and z , which are useful objects

CHECKLIST FOR RECURSIVE ALGORITHMS

begin

if($\langle a, b, c \rangle$ is a sufficiently small instance) return($\langle 0, 0, 0 \rangle$)

$\langle a_{sub1}, b_{sub1}, c_{sub1} \rangle =$ a part of $\langle a, b, c \rangle$

$\langle x_{sub1}, y_{sub1}, z_{sub1} \rangle = Alg(\langle a_{sub1}, b_{sub1}, c_{sub1} \rangle)$

$\langle a_{sub2}, b_{sub2}, c_{sub2} \rangle =$ a different part of $\langle a, b, c \rangle$

$\langle x_{sub2}, y_{sub2}, z_{sub2} \rangle = Alg(\langle a_{sub2}, b_{sub2}, c_{sub2} \rangle)$

$\langle x, y, z \rangle =$ combine $\langle x_{sub1}, y_{sub1}, z_{sub1} \rangle$ and $\langle x_{sub2}, y_{sub2}, z_{sub2} \rangle$

return($\langle x, y, z \rangle$)

end algorithm

CHECKLIST FOR RECURSIVE ALGORITHMS

- 1) **Specifications:** You must clearly define what the algorithm is supposed to do.
- 2) **Variables:** It is also important to carefully check that you give variables values of the correct type, e.g., k is an integer, G is a graph, and so on.
- i) **Input:** The first line of your code, algorithm $Alg(a, b, c)$, specifies both the name Alg of the routine and the names of its inputs. Here a, b, c is the input instance that you need to find a solution for
- ii) **Output:** You must return a solution x, y, z to your instance a, b, c through a return statement $return(x, y, z)$.

CHECKLIST FOR RECURSIVE ALGORITHMS

- **Every path:** if your code has *if* or *loop* statements, then every path through the code must end with a return statement
- **Type of Output:** Each return statement must return a solution x, y, z of the right type
- **Few Local Variable:** An iterative algorithm consists of a big loop with a set of local variables holding the current state. Each iteration these variables get updated.
- **3) Tasks to complete:** Your mission, given an arbitrary instance a, b, c meeting the preconditions, is to construct and return a solution x, y, z that meets the post condition.