**AALIM MUHAMMED SALEGH COLLEGE OF ENGINEERING**

**DEPARTMENT OF COMPUTER APPLICATION**

**SEMESTER- I**

**MC5102 – PROBLEM SOLVING AND PROGRAMMING**

**UNIT – II NOTES**

**SYLLABUS:**

**UNIT I INTRODUCTION TO COMPUTER PROBLEM SOLVING**

Introduction – The Problem Solving aspect – Top down design – Implementation of algorithm – Program Verification – The efficiency of algorithm – The analysis of algorithm.

**Programs and requirements for problem solving**

Problem solving can be stated as intricate process requiring much thought, careful planning, logical precision, persistence and attention to detail. It can be challenging, exciting and satisfying experience with considerable room for personal creativity and expression.

**Programs and Algorithms**

Computer solution to a problem is a set of explicit and unambiguous instructions expressed in a programming language.

**Set of instructions is called a program.**

A Program can thought as algorithm expressed in a programming language.

An algorithm corresponds to a solution to a problem that is independent of any programming language.

Program should supply with input data.

Input data manipulate according to the instructions and produce the output.

Output represents computer solution to the problem.

Problem



Algorithm



I/P  Computer O/P

**Algorithm:**

Algorithm consists of a set of explicit and unambiguous final steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in a finite time.

**Requirements for solving problems by Computer**

One can employ algorithms to solve problems.

Depth understanding is needed to design algorithm which solve any complex problem.

**Example : Telephone Directory Look up Problem:**

A Telephone directory has thousands of names and numbers.

Searching by name or number from page 1 is lengthy and time consuming process.

Probably the directory is sorted by number than name.

Data structure is linked with algorithm for high performance.

**Problem Solving Aspect**

Problem Solving is a creative process which largely defines systemization and mechanization.

Problem solving has number of steps to achieve the goal.

**Problem definition phase:**

Defining the problem fully is done in the problem definition phase.

This phase decides “What must be done” rather “How to do it”.

Problem statement gives a set of precisely defined tasks.

**Getting started on a problem:**

Many ways to solve most problems.

Many solutions to most problems.

Difficult to identify which path is fruitful and fruitless while solving a problem.

After getting complete idea it is better to start implementation. It means “What can we do”.

Proverb states that “Sooner your start coding your program the longer it is going to take”.

**Use of specific examples:**

Many strategies, while using, we may stuck in some point.

It is usually much easier to work out the details of a solution to a specific problem.

Geometrical or schematic diagrams representing certain aspects of the problem can be usefully employed in many instances.

**Example:** The Greatest common divisor of 2 numbers.

**Problem:** Given 2 positive non zero integers n and m. find GCD of n and m.

**Algorithm development:**

**Ordinary Procedure**

Find the common divisors of both n and m.

hen find the GCD of both in common.

**Step 1:** Find the prime factors of m

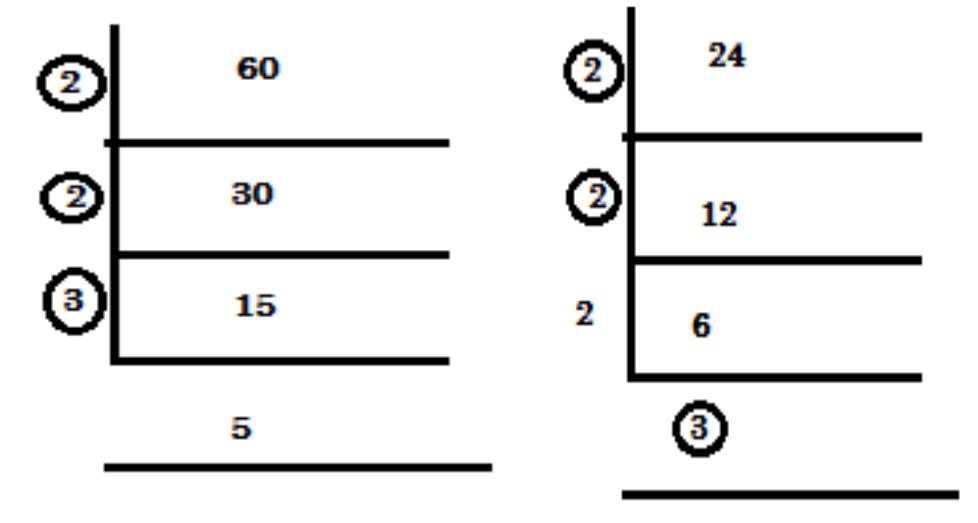
**Step 2:** Find the prime factors of n

**Step 3:** Find the common factors in prime expansion

**Step 4:** Compute the product of common factors. It gives the output.

**Example :** input m and n

Say m=60 and n=24



**2×2×3=12**

* 1. GCD of 60, 24 is 12. Here step 3 is tedious.

**Euclid’s Algorithm:** gcd(m, n) = gcd(n, m mod n)

i.e Remainder of division m by n

**Step 1:** If n=0 return value of m and stopElse goto step 2

**Step 2:** Divid

**Step 3:** Assign m=n and n=r

Goto step 1 i.e making r=0.

**Pseudo code:**

While n≠0

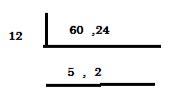
r←m mod n

m←n

n←r

return m

Thus, largest integer that divides both numbers evenly is called GCD.



**Another Method:**

The GCD of 2 numbers may be the least of the 2 numbers (or) less than the small number of the given 2 numbers

Take min of 2 numbers t = min {m,n}

Check whether t divides m and n

if (t)

Then t is the answer

Else

T=t-1

Decrease by 1

**Algorithm:**

Step 1: t=min {m,n}

Step 2: Divide m by t

M mod t = 0

Goto step 3

Other

Goto step 4

Step 3: Divide n by t

If (n mod t = 0)

Return t

Else

Goto step 4

Step 4: t = t-1

Goto step 2

Algorithm work fine when one i/p is zero. Otherwise it is time consuming.

**Similarities among problems:**

Get the past experience for any current problem

Start any solving process with specific example.

Have aware about the similarities of various problems.

More experience in more tools and techniques helps to solve the problem easily.

Sometimes previous experience blocks new creativity or better solution.

Place only cautious reliance on past experience.

Independently solving the problem sometimes gives better solution.

Analyzing past problems and experience sometimes leads to dead end.

Solve a problem after viewing the problem from different angles.

Analyze the problem by turning a problem upside down, inside out, sideways, backwards, forwards and so on…..

**Working backwards from the solution**

Try to work backwards to the starting conditions. This is significantly better.

Important thing in developing problem solving skills is practice.

Piaget says that “We learn more when we have to intent”.

**General problem solving strategies:**

There are many general and powerful computational strategies that are used in many guises in computer science.

Mostly used principle is “Divide and Conquer Strategy”.

Divide and Conquer:

It divide the original problem in to sub problems

It is possible to split the problem into smaller and smaller sub problems.

Applied for sorting, selecting and searching algorithms.

**Example** : Binary search algorithm

**Step 1:** Arrange the elements in order (sequential order)

**Step 2:** Split the list and find the middle value and compare with theelement to be searched.

**Step 3:** If middle value > element to be searched

Perform step 2 for second half

Else

Perform step 2 for first half

Perform step 2 and 3 until goal achieved.

A[0]………A[m-1] A[m] A[m+1] …A[n-1]. Find k l = 0, r = n-1

**Step 4:** Split the sorted array (l+r)/2

**Step 5:** if k = A[m] m-> middle index

Stop

Elseif

Divide and conquer for k < A[m]

Else

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Divide and conquer applied k > A[m] (second half of the list). | | | | | | |
| 3 | 14 | 27 | 31 | 39 | 42 | **55** | **m** |

****

70 74 81 85 93 98

**r**

Need log2n comparison rather than n comparison.

It is good for searching problems.

**Example:** Dynamic Programming

Build up a solution to a problem through a sequence of intermediate steps.

Idea that a good solution to a large problems by finding good solution to smaller problems

It is technique for solving problems with overlapping sub problems

**Example**: Fibonacci series.

**0** **1** **1** **2** **3** **5** **8** **13** **21** **34**

**By Recurrence:**

**F(n) = f(n-1) + f(n-2) for n≥2**

Initial condition f(0)=0 f(1)=1

n=5

**Dynamic Programming algorithm:**

Refined to avoid using extra space.

Avoid solving unnecessary sub problems.

It is an algorithm design technique for optimization problems.

Solves problems by combing solutions to sub problems.

Sub problems dependent.

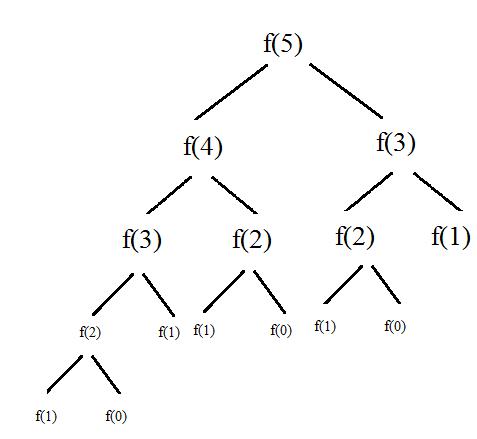
Solutions of sub problem not affect another sub problem.

**Top Down Design**

An algorithm is able to implement a correct and efficient computer program**.**

To solve a problem powerful techniques are used to design an algorithm.

The problem can be solved effectively if the algorithm manages the inherent complexity.

A technique for algorithm design that tries to accommodate some human limitations is known as Top down design or stepwise refinement.

Solve the problem in stepwise fashion.

**Breaking a problem into sub problems**

First the ground work should be done that gives the outline of a solution.

Problem description itself sometimes gives starting point for top down desi gn.

Outline may have set of statement or a single statement.

One statement or task can be splitted into sub tasks.

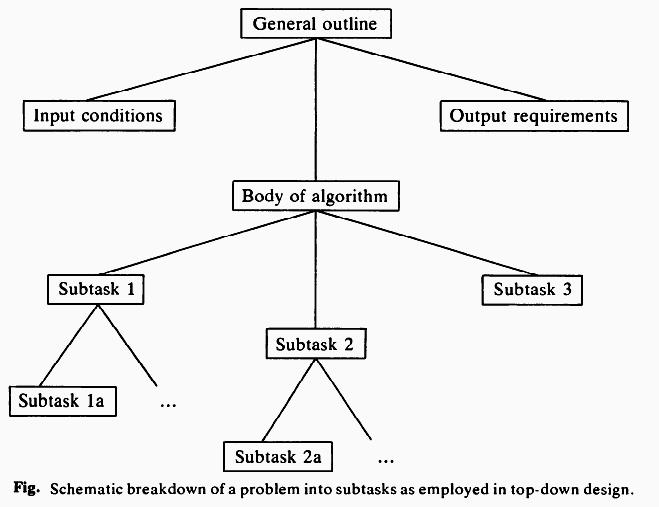
These well-defined sub tasks are useful to reach the final goal.

Sub tasks need to interact with each other should well defined. This preserve over all structure of the solution to t he problem.

Preservation of overall structure needed to prove the correctness of the sol ution.

Large task is divided into sub tasks. Sub tasks are again sub divided into sub tasks. End sub tasks form the program statement.

Schematic breakdown of a problem into sub tasks as employed in top dow n design.



**[2] Choice of a suitable data structure**

* 1. Important decision in formulating computer solutions to problems is the choice of appropriate data structures.
  2. Organized data has the effect on final solution.
  3. Inappropriate choice leads to a simple, transparent and efficient implementation.
  4. Data structure can be defined at any stage of the algorithm development.
  5. It is desirable when considered DS at very outset of our problem solving explorations before the top down design.
  6. It can also be refined as the algorithm is developed.
  7. Data structures and algorithms are linked to each other.
  8. A small change in data organization can have a significant influence on an algorithm.
  9. There is no formula that is problem must use this choice of data structure.
  10. Things to be asked or aware when DS chosen are
      1. How can the intermediate results are arranged that reduce computation in the later stage?
      2. Can the data structure be easily searched?
      3. Can the data structure be easily updated?
      4. Can earlier state can be recovered?
      5. Whether need excess storage?
      6. It is possible to impose some data structures on a problem that is not initially apparent?
      7. Can be problem be formulated in terms of one of the common data structures. Example: array, set, queue, stack, tree, graph, list?

**Construction of loops:**

Sub tasks are leads to series of iterative constructs, loops or structures that work under condition.

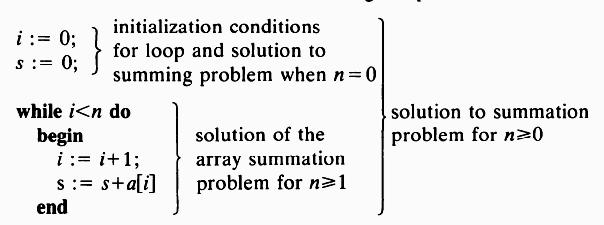
These, work with i/p or o/p statements, computable expressions and assignments form the heart of the program implementation.

Loop has three important parts

* + - 1. Initial condition – loop begins
      2. Invariant relation – apply for each iteration
      3. Terminate (condition) – Under which iterative process work or terminate.

Some problems use straight forward process instead of loops.

**Summation problem**

****

1. **Establishing initial conditions for loops** Set loop variable to a value
   1. The range may be 0≤i≤n for n iterations.
   2. A smallest problem has i=0 or i=1
   3. n=0 means i=0 and s=0
2. **Finding the iterative construc t**
   1. To solve a problem of i=1 th en we must solve i=0 The solution for n=1 is



i=1

s=a[ 1]

The solution for n>1 is



i=i+1

s=s+a [i]

The solution to summation problem is n≥0



i=0; s=0;

while i <n do

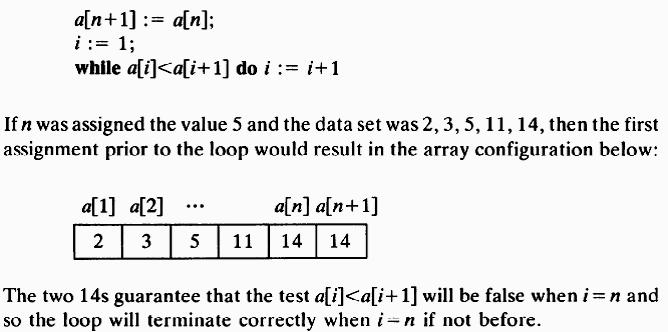
beg in

i=i +1;

s=s+a[i];

en d

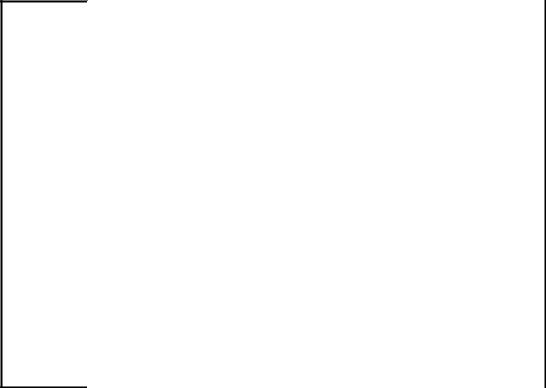
Example 3 with sample data



**Termination of loops** Loops terminated by

* + 1. Known number of it erations.
    2. Direct termination fa cility in program code.

**Example 1:** Pascal for l oop



For i=1 to n do

Begin

.

.

.

end

Unconditionally loop terminates after n

**Example 2:** Pascal while loop



While(x>0) and (x<10) do

begin

.

end

No guarantee of termination of loop

**Example 3:** Termination possible with simplifying test.

Establish an array of n elements

a[1]<a[2]<……<a[n]

a[n+1]=a[n]

i=1

while a[0]<a[i+1]

do

i=i+1;

**IV.** **Implementation of Algorithms**

* 1. Properly designed in a top down fashion
  2. Top down rule easy to understand and debug
  3. Also easy to modify because the program are much more apparent.

1. **Use of procedure to emphasize modularity:**
   1. Top down design are easy to understand and more readable, thus it can be modularize.
   2. Modularize means splitting the large or lengthy program into set of independent procedures.
   3. This set of procedures will perform well defined tasks.

Procedure sort

Begin

Writeln(‘sort called’)

end

**Choice of variable names:**

Proper variable names are chosen so that easy to understand and remember in future work.

**Example :** To name a variable to store a day means, variable name is day instead of justd or a.

A clean definition of all variables and constants at the start of each procedure can makes the program more meaningful. This is a process of self documenting.

**Documentation of programs:**

A program should be more effective and useful if it understood and used by other people.

It must give the extract requirement i.e input with format for the user.

**Example:** Enter a number between 0 to 10 as a whole number.

The program should prope rly handle the wrong request i.e proper resp onse should be given.

**Debugging programs:**

Various test should be done even for a small program to have a error free program.

Additional information give n with output that it can work normal or abnormal situation respect to the input given.

A simple debugging tool is Boolean variable.

**if debug the n**

**Begin**

**Writeln(…..)**

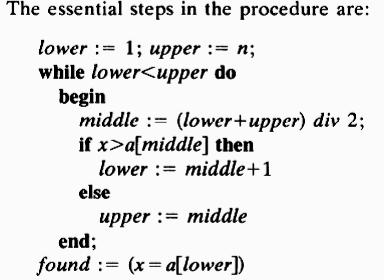
**End**

Error should be handled more effectively which results in a good program.

Work the program by hand before put in to the system.

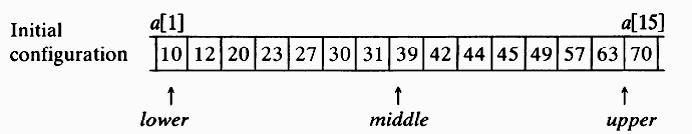
No method for debugging b ut some steps makes the task easy.

Example : Binary search pro cedure

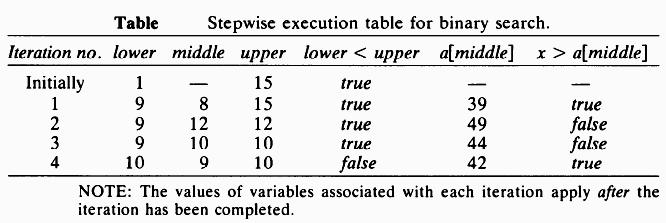


**Problem**

1. Search value x
2. Array a[1….n]
3. x=44; n=15



1. To get proper result do the task by hand written stepwise procedure.
2. Do not assume anything. M ake it clear and then proceed.



**Program Testing**

A program must be designed to solve a problem which can accommodate with limiting and unusual cases.

Unusual cases make the program critical.

Necessary to handle all types of inputs.

A program should properly respond to the user inputs.

Writing a program to a specific case generally need a lot of time and effort.

Program must solve wide area of problems.

Instead of fixed constants, variables are used.

**Example 1:** while i<100 do // Fixed constatnt

**Example 2:** while i<n do // Variable

Fixed constant are useful for specific cases. Example Tmonths=12;

**Program Verification**

Software development an d debugging need more time and effort.

Large program need more time and more effort.

Top down design are useful to make the program readable and understandable.

Program correctness can be a matter of life or death in the case of military, space and medical applications.

Demonstrating program correctness is more needed that working different input to a particular problem.

**Program Verification:**

It refers to a application of mathematical proof techniques to establish that the results obtained by the execution of a program with arbitrary inputs are in accord with formally defined output specifications.

Prove the algorithm at the basic level itself or abstract or superficial level.

**Computer model for program execution**

A program may have variety of path for termination.

According to the input path has chosen for execution and terminate.

A program may transit from one state to another.

A state transition changes the value of the variable in a current execution path.

As well as instructions that change the computation state there also other instructions that simply makes tests on the current state.

These tests make change in the sequential flow of execution.

This model for program execution provides us with a foundation on which to construct correctness proofs of algorithms.

**Input and Output assertions:**

Program correctness depends on formal statement

Formal statement has 2 parts:

Input assertion

Output assertion

**Input assertion: Specify any condition placed on the values of the input variable**

**Output assertion: Produce for input data that satisfies the input assertion.** (x=q \* y+r) ^ (r<y)

The output assertion can written as logical notation

(x=q \* y + r) ^ r<y

- Logical connectivity “and” Q - Quotient

R - remainder

x divided y

**Implication and Symbolic execution**

Problem can be verified by a set of implication.

General form of implication

P  Q

P 🡪 assumption

Q 🡪 conclusion

|  |  |  |
| --- | --- | --- |
| P | Q | PQ |
| True | True | True |
| True | False | False |
| False | True | True |
| False | False | True |

If assumption and conclusion are same then true else if conclusion is true then true.

Symbolic execution replaces all input data values into symbolic value and all arithmetic operations into algebraic manipulation of symbolic expressions.

Normal execution

x=3 y=1

x=x-y ===> x=3-1

=2

Symbolic execution

x=α y=β

x=x-y ===> x=α-β

if x=α-β and y=β

then find y=x+y

y= α-β+ β

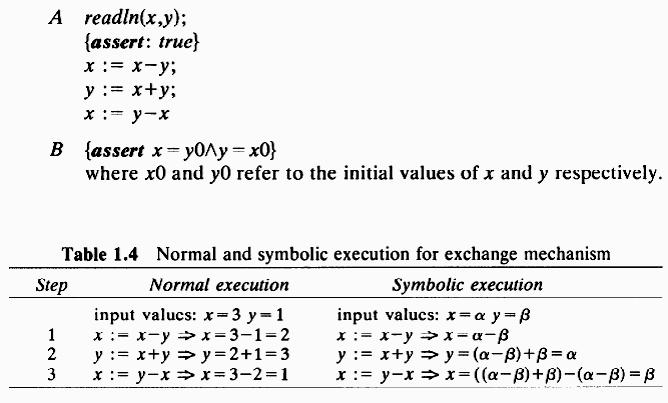
y= α

Symbolic execution enables us to transform the verification procedure into proving that the input assertion with symbolic values substituted for all input variables implies the output assertion with final symbolic values substituted for all variables. This is called as Verification condition.

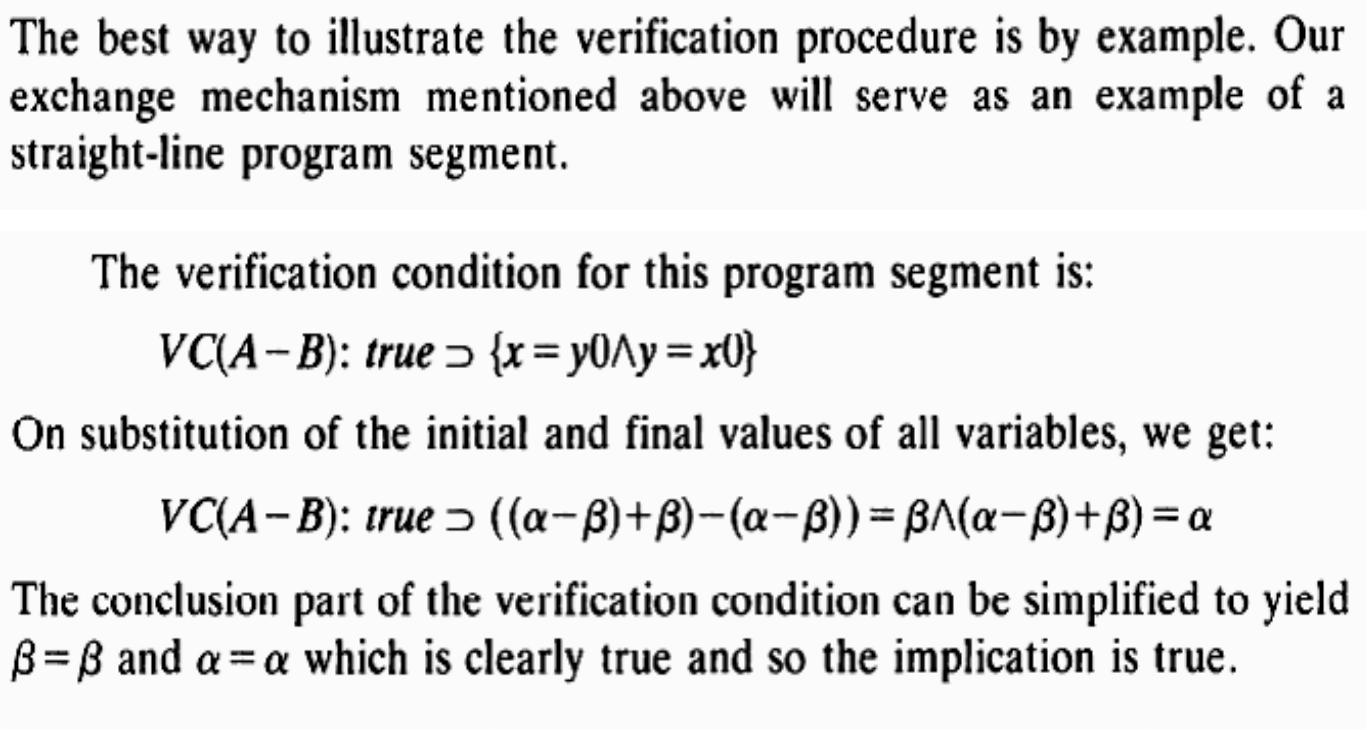


A number of intermediate verification conditions between the input and o utput assertions are needed.

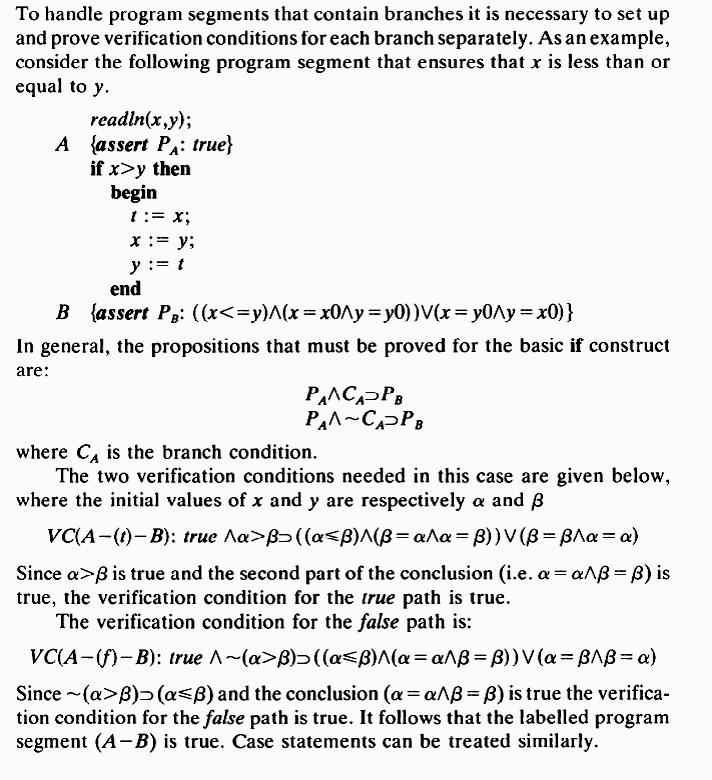
Verification conditions are straight line segment, branching segments and loop segment.



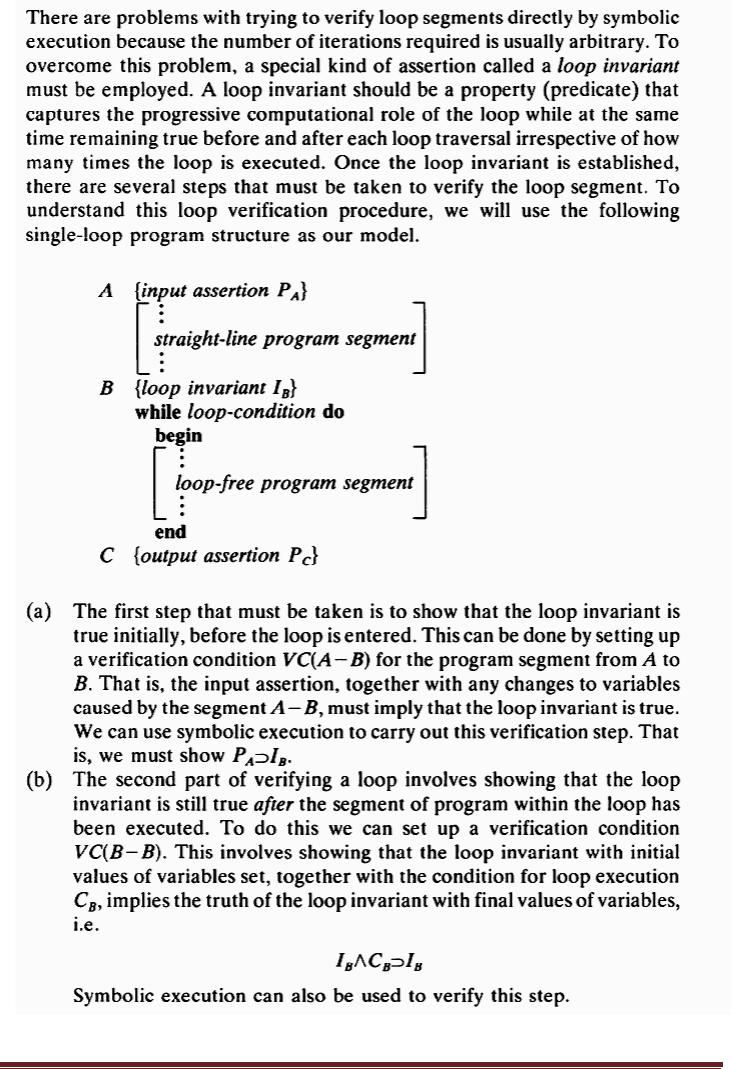
**[4] Verification of straight line program segments:**

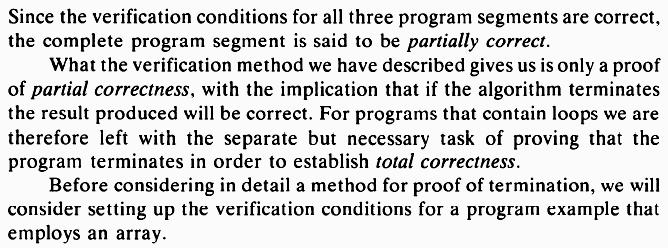
****

**[5] Verification of program segm ents with branches:**

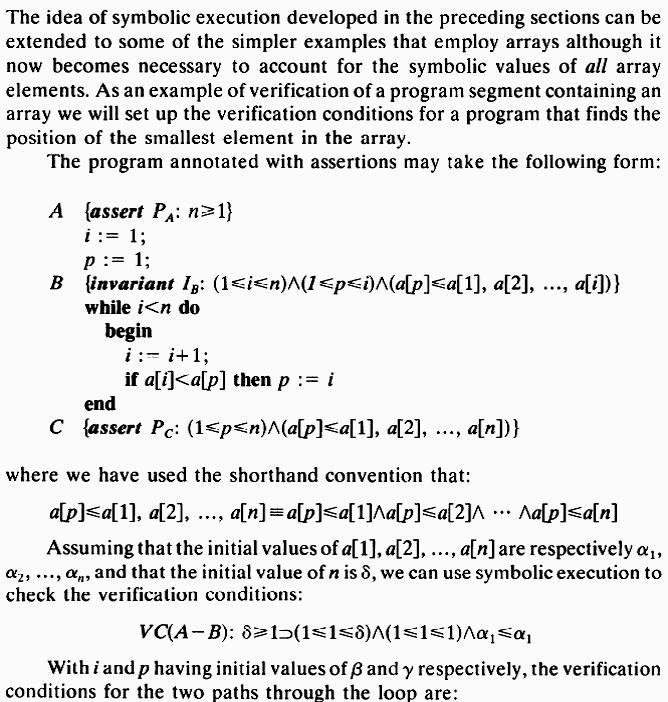
****

**[6] Verification of program segm ents with loops:**

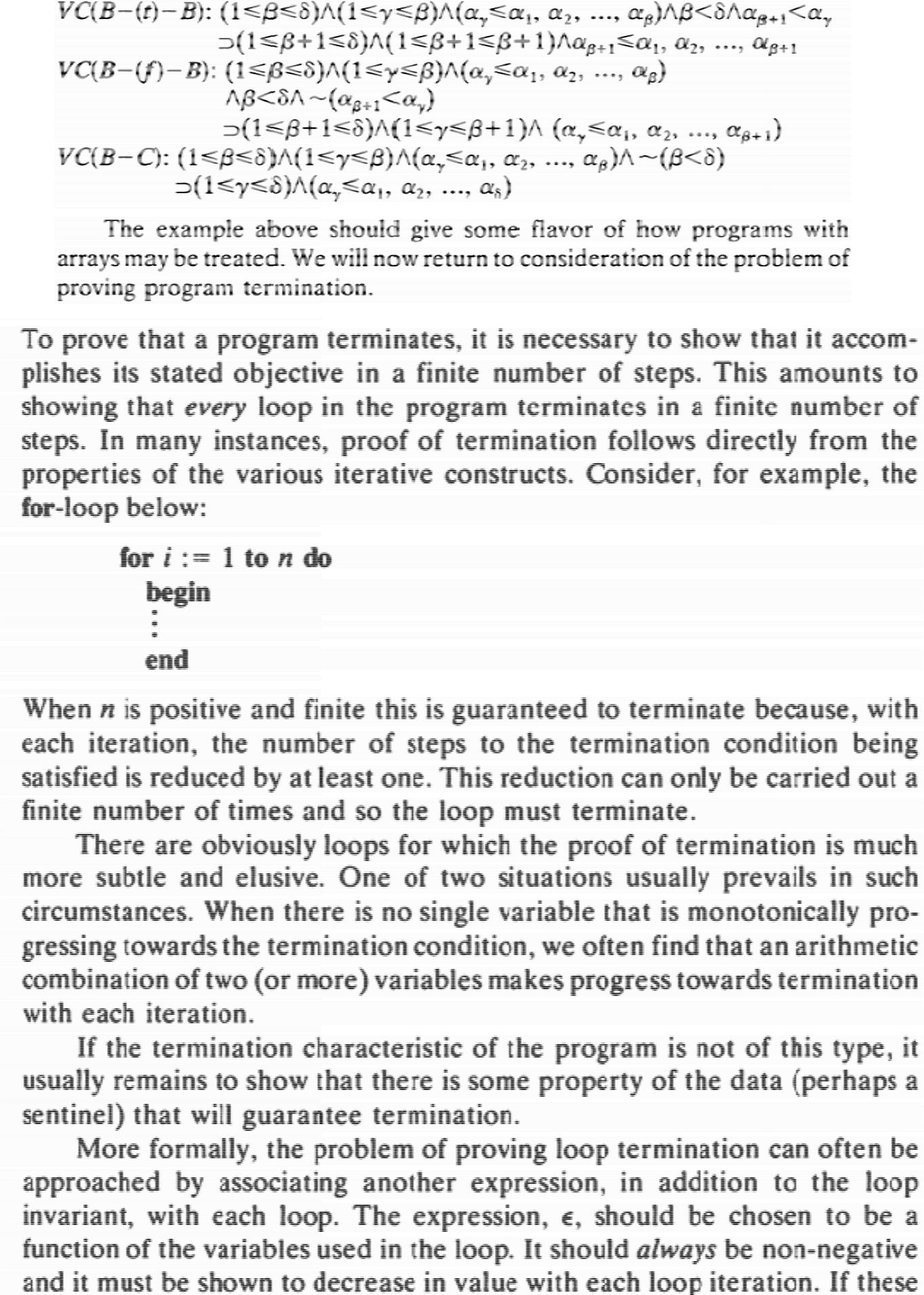


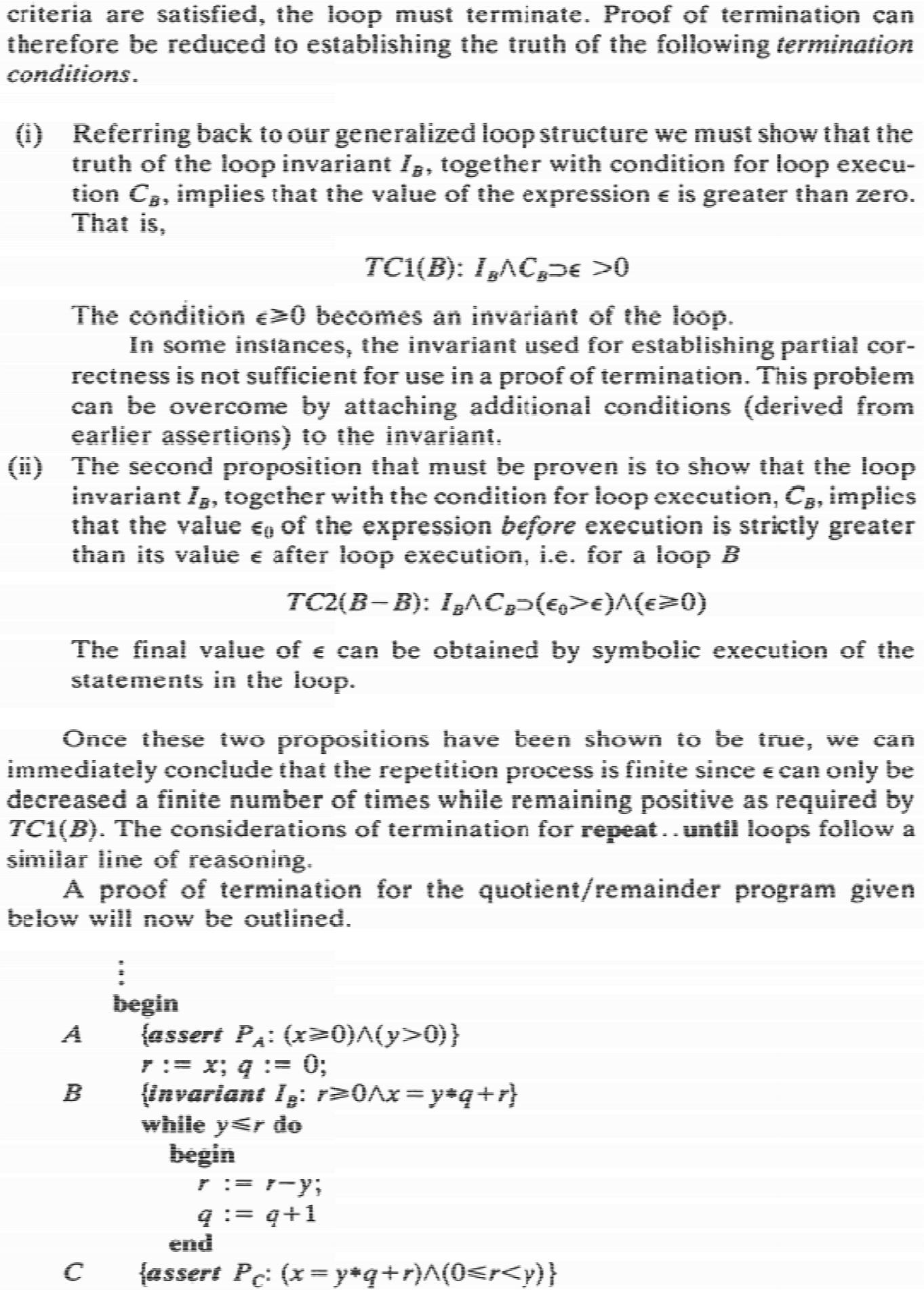


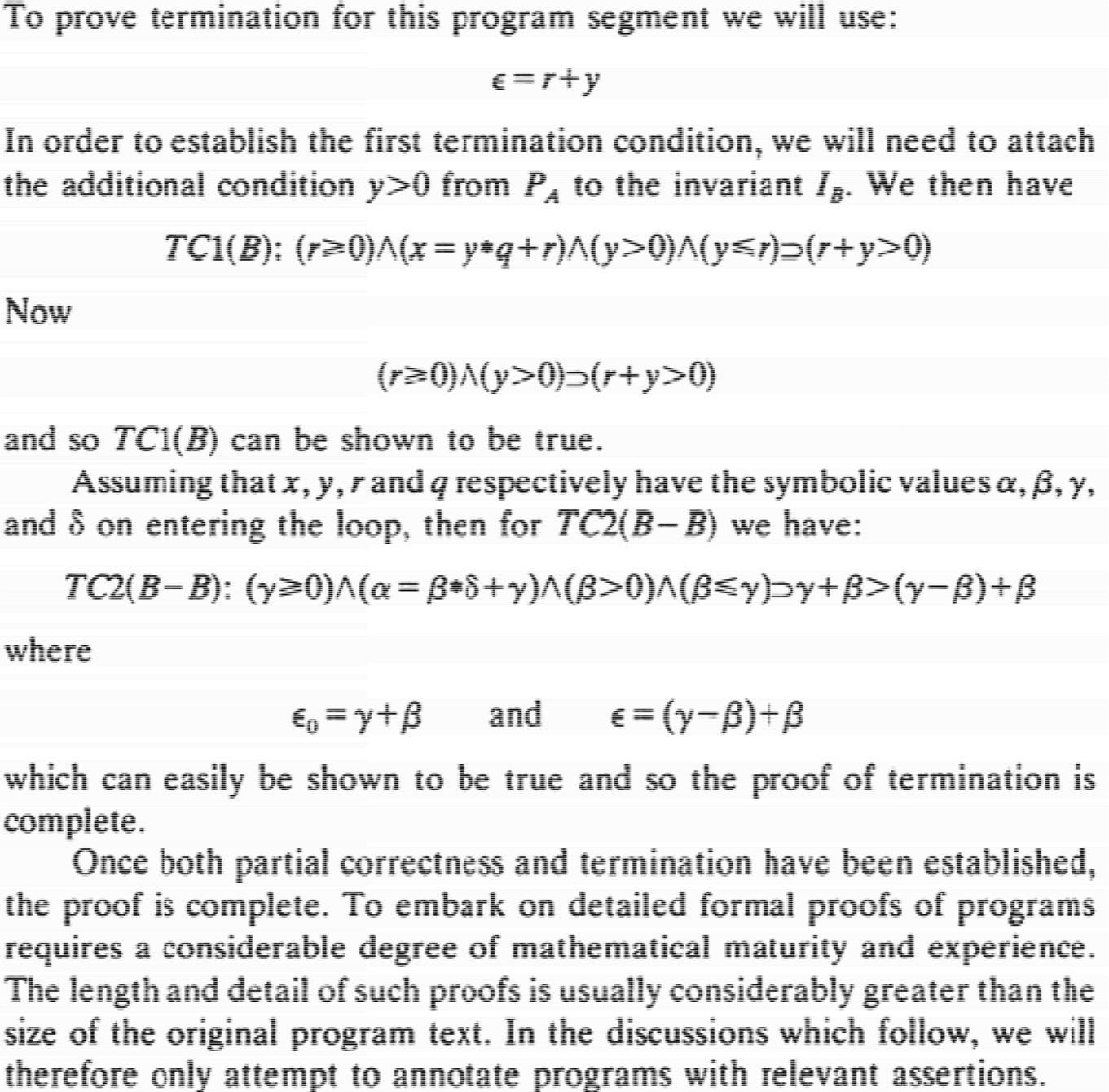
**[7]Verification of program segments that employ arrays**









**VI.** **The efficiency of algorithms**

Efficiency lies on design, im plementation and analysis of algorithms

CPU and internal memory efficiency helps to improve algorithm efficiency .

Computer resources are nee ded to complete the task of a algorithm.

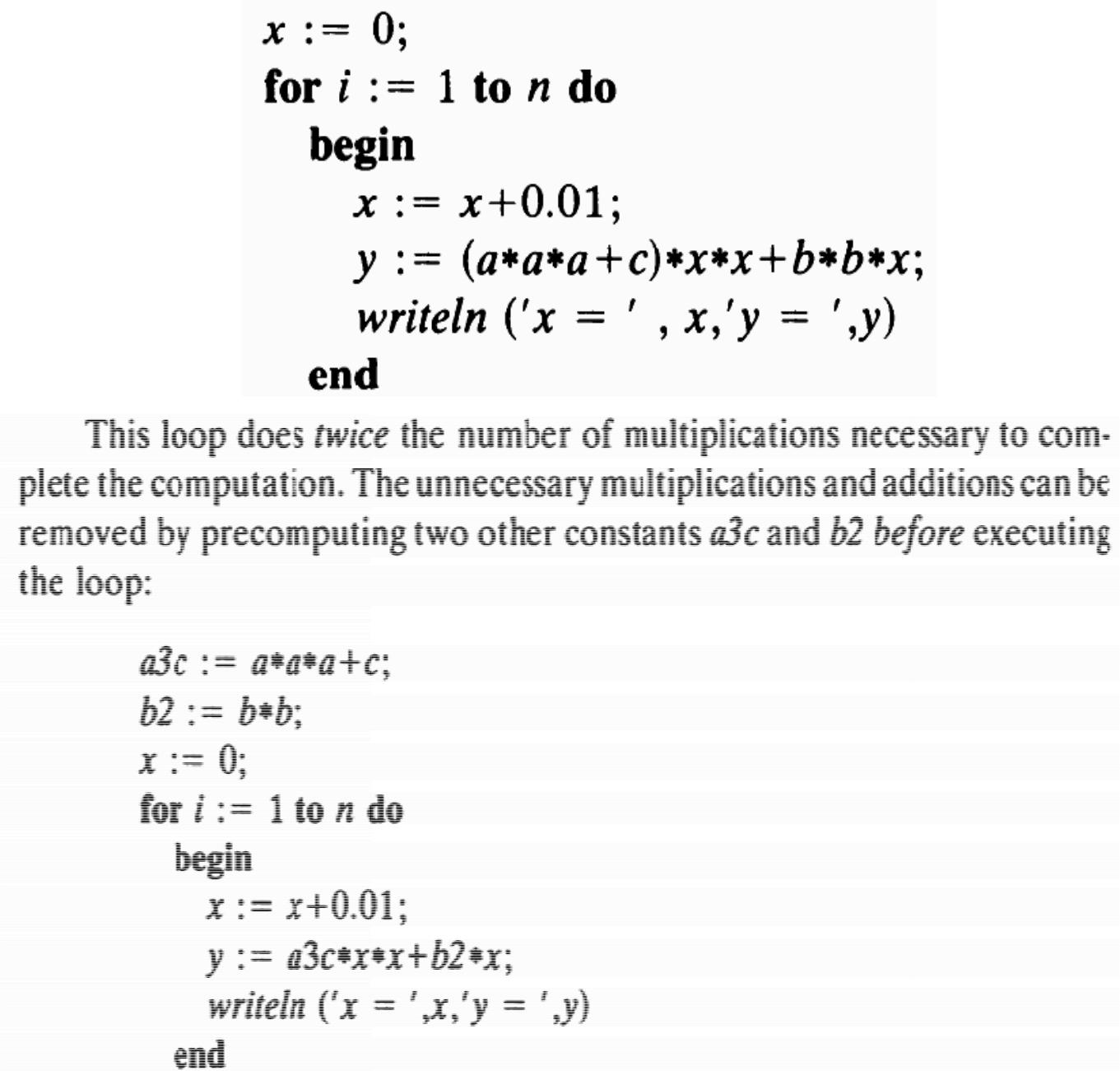
Always aware to design a algorithm which was more economical.

This is possible by only by giving specific response regarding to the char acteristics of a problem.

1. **Redundant Computations:**

Redundant computation leads to inefficiency.

1. When it occurs inside for loop or any other loops, it will be executed m any times. That leads more serious.
2. Repeatly recalculating s ame set of statements remains constant.
3. Unnecessary multiplications and additions should be removed.



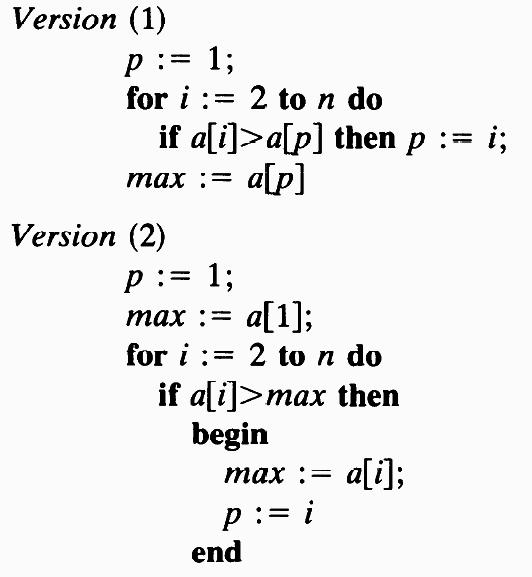
Redundancy inside the inner loops should be eliminated.

1. **Referring array elements:**

Redundancy easily creeps into array processing.

Version 1 is not more efficient because of condition a[i]>a[p].

The condition a[i]>a[p] need only one memory reference.



1. **Inefficiency due to late termination**

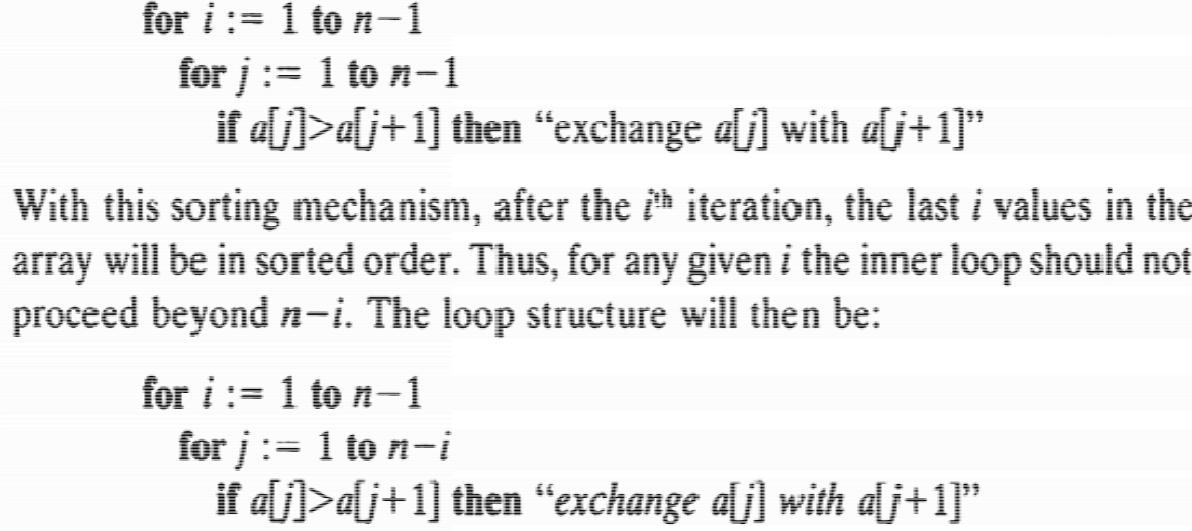
The inefficiency will occur if the algorithm terminates late.

1. Late in-sense, suppose after achieving the goal the other data also visited.
2. Example: Alphabetical (linear) search

In linear search each data is visited and if target achieved the program term inates. If it does not terminate then occur inefficiency.

Version 1 terminates after visiting complete list of items.

Version 2 terminates once the target is obtained.



**Early detection of desired output conditions**

Inefficiency sometimes involved due to early detection of desired output conditions.

This early detection of desired output condition leads to termination problem Eg Bubble sort

Due to the nature of input the output condition met early before termination condition met.

This will happen when i/p list is in sorted order.

**Trading storage for efficiency gains**

To speed up an algorithm, always include least number of loops.

One loop to do one job is better. This is like one variable hold one value.

Trade between storage and efficiency improve performance.

Save some intermediate results to avoid unnecessary testing.

Always use less memory space algorithm to improve efficiency.

**VII.** **The Analysis of algorithms**

Every one follow algorithm which gives good solution.

Good solution to a problem is always appreciable.

Good solution should be quantitative or qualitative.

The solution should be more economical.

Economical in terms of human as well as system.

**Good algorithm qualities and capabilities**

They are simple but powerful and general solution.

Easy to understood and clear in implementation without tricky.

Easily modified if needed.

Correct for clearly defined situations.

Able to understand on number of levels.

Economical in terms of time, storage and peripherals.

Documented clearly that anyone can understand.

Must machine independent.

Used as sub procedure for other problems.

The solution must please, satisfy and made to feel proud to its designer. Quantitative measures give the way to predict the performance of an algorithm and can be comparing with other algorithm of same problem.

A algorithm is efficient if its saves computing resources which saves times and money.

**[1]Computational complexity:**

Computational model that gives the algorithm performance foe specified input conditions.

1. It is a quantitative measurement.
2. Performance measured in terms of problem size (n).
3. n increases then cost increases.
4. Lower end of the scale, also have logarithmic dependence of n.
5. Higher end of the scale, also have exponential dependence on n.

**Computational cost as a function of problem size for a range of computational complexities.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Log 2n** | **N** | **Nlog2n** | **N2** | **N3** | **2n** |
| 1 | 2 | 2 | 4 | 8 | 4 |
| 3.322 | 10 | 33.22 | 102 | 103 | >103 |
| 6.644 | 102 | 664.4 | 104 | 106 | >>1025 |
| 9.966 | 103 | 9966.0 | 106 | 109 | >>10250 |
| 13.287 | 104 | 132,877 | 108 | 1012 | >>102500 |

* 1. One can solve only very small problem with an algorithm that exhibits exponential behavior.
  2. Logarithmic dependence on n

If problem n=104, 13 steps needed 13 micro seconds needed.

* 1. Exponential algorithm on n if n=100, Time taken : Earth terminate
  2. N grows due to comparison or number of times some arithmetic expressions repeated.

**The Order notation**

A standard notation developed to represent functions which bound the computing time for algorithms.

It is three types.

Usually O notation is used. O notation can be called “Big O” notation.

An algorithm in which the dominant mechanism is executed cn2 times for c, a constant and n problem size is said to be order n2 complexity. It can be written as O(n2).

A function g(n) is Of(n) provided there is a constant c, the relation is g(n) ≤ c f(n) holds for all value of n that are finite andlimpositive. =

G(n) and f(n) can be expressed as:→:

C is not equal to zero.

Example:

An algorithm that requires 3n2+6n+3 comparisons to complete its task.

1. (**G(n)=**)= **3n2+6n+3**
   1. **3n2+6n+3 =3**

**N2**

The particular algorithm has an asymptotic complexity of O(N2).

An algorithm with a higher asymptotic complexity has a very small constant of proportionality and hence for some particular range of n it will give better performance than an algorithm with lower complexity and a higher proportionality constant.

Worst and average case behavior

Worst and average case applied to both the time and space complexity of an algorithm.

Worst complexity:

Given problem size n corresponds to the maximum complexity encountered among all problem of size n.

In many practical applications it is much more important to have a measure of the expected complexity of a given algorithm rather than the worst case behavior.

The expected complexity gives a measure of the behavior of the algorithm averaged over all possible problems of size n.

In comparing 2 algorithms to solve a given problem, generally opt in preference for the algorithm that has the lower expected complexity.

**Probabilistic average case analysis**

* 1. Characteristic the behavior of an algorithm that linearly searches an ordered list of elements for some value x.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | ………… | N |

Worst case:

Necessary for the algorithm to examine all n values in the list before terminating.

Average case:

A probabilistic average case analysis it is generally assumed that all possible points are equally likely, i.e the probability that x will be found at position 1 is 1/n and at position 2 is 1/n and so on.

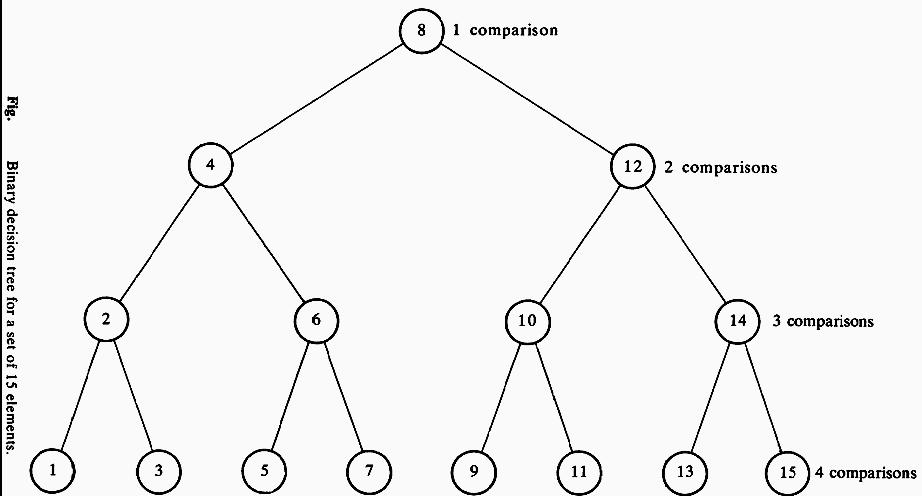
1. The average search cost is t herefore the sum of all possible search costs ea ch multiplied by their associated probabili ty.

Example: If n=5

Average search cost=1/5(1+ 2+3+4+5) = 3 General case

Average search cost=1/n(n/ 2(n+1)) =n+1/2

1. Average number of iteration s of the search loop that are required before th e algorithm terminates in a successful se arch.
2. The associated binary searc h for an array of size 15.



* 1. One element can found with 1 comparison
  2. Two elements with 2 comparisons.
  3. Four elements with 3 comparisons
  4. So on……

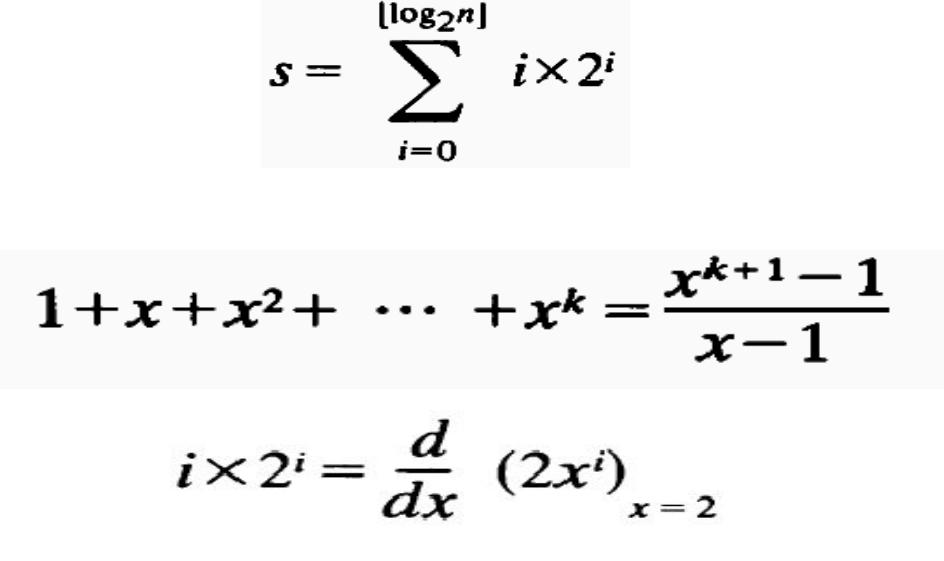
The sum of over all possible elements = 1+2+2+3+3+3+3+4+… ..

* 1. 2i elements require i+1 comparison.

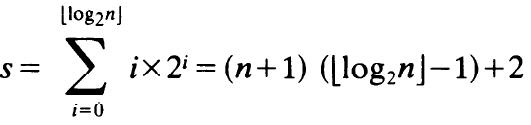
Average search cost is again just the sum over all possible search costs, each multiplied by their associated probability.

1. It is exact only when n is one l ess than a power of two. Some calculus is nee ded to evaluate the sum

Sum is a geometric progress ion



1. To compute the sum take the derivation of (xk+1)/(x-1) multiplied by 2
2. After substituting the sum in average search cost expression, average search cost is got



**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***