Module – 1

Dr. P. Gayathri

Professor

SCOPE, VIT Vellore

Algorithm

- An Algorithm is a set of rules for carrying out calculation either by hand or on a machine
- **An Algorithm** is a well defined computational procedure that takes input and produces output
- An Algorithm is a finite sequence of instructions or steps to achieve some particular output

Algorithm

- Any Algorithm must satisfy the following criteria
  - **Input:** It generally requires finite no. of inputs
  - **Output:** It must produce at least one output
  - **Uniqueness:** Each instruction should be clear and unambiguous
  - **Finiteness:** It must terminate after a finite no. of steps

Algorithm

- Analysis issues of an algorithm
  - What data structures to use !!!
  - Is it correct ???
  - How efficient is it ???
  - Is there any other efficient algorithm ???

Algorithm

- An algorithm can be viewed as tool for solving a well-specified **computational problem**
- An algorithm is said to be **correct** if, for every input instance, it
  - halts with the correct output
- An incorrect algorithm
  - may not halt at all for some input instances, or
  - may halt with a wrong output

Algorithm

- A computer program is written in a programming language whereas an algorithm is written in pseudo code.

Stages of Algorithm Development

- Describe the problem – define the problem stmt.
- Identify a suitable technique
- Design an algorithm
- Check the correctness of the algorithm – works fine for all selected inputs.
- Measure the time complexity of the algorithm

Ex: Swapping of the values of two variables

- Problem Description:
  - Given two variables, a and b, exchange the values assigned to them.
- Suitable technique: Usage of a temporary variable
- Design an algorithm
- Check the correctness by sample input and output
- Measure the time complexity of the algorithm

Ex: Swapping of the values of two variables

Sample Input: Before exchange

a	b
135	245

Desired Output: After exchange

a	b
245	135

- Applications
  - Sorting Algorithms

Algorithm Design

Algorithm: EXCHANGE VALUES OF TWO VARIABLES

1. Save the original value of a in t.
2. Assign to a the original value of b.
3. Assign to b the original value of a that is stored in t.

Algorithm (pseudo code format): EXCHANGE VALUES OF TWO VARIABLES

1. t ← a
2. a ← b
3. b ← t

Proof of Correctness of the algorithm

- Once an algorithm is designed, we need to prove its correctness i.e. the algorithm yields the desired output for the specific input in a finite amount of time.
- Proof of correctness depends on the algorithm.
- For some algorithms, the proof of correctness is easy and for some algorithms, it is complex
- It depends on the type of instructions in the algorithm

Proof of Correctness of the algorithm

- In order to show that an algorithm is incorrect, one instance of input is sufficient for which the algorithm fails to give the desired output
- For the swapping of the values of two variables, proof of correctness is straightforward i.e., we can check the pre-condition and post-condition of the values of the variables

Proof of correctness - Swapping of the values of two variables

Precondition:

a	b
135	245

Post condition:

a	b
245	135

Proof of correctness - Loop invariants

- Loop invariants - means checking the status of the control variables in the loop
- 3 things related to loop invariant
  - Initialization: Check whether the invariants used in the loop are properly initialized such that the condition(s) are satisfied to enter the loop i.e., it is true prior to the first iteration of the loop

Proof of correctness - Loop invariants

- Maintenance: Check whether the values of the invariants are updated and progress towards the termination of the loop i.e., if it is true before an iteration of the loop, it remains true before the next iteration
- Termination: When the loop terminates, the invariant gives a useful property that helps to show that the algorithm is correct

Proof Correctness - Summation of a set of n numbers

Problem: Given a set of n numbers, design an algorithm that adds these numbers and returns the resultant sum.

Algorithm: SUMMATION

Input: array of n numbers

Output: sum of the n numbers

sum← 0

k ← 0

While(k< n)

do sum ← sum + a[k]

k ← k + 1

return sum

Proof Correctness - Summation of a set of n numbers

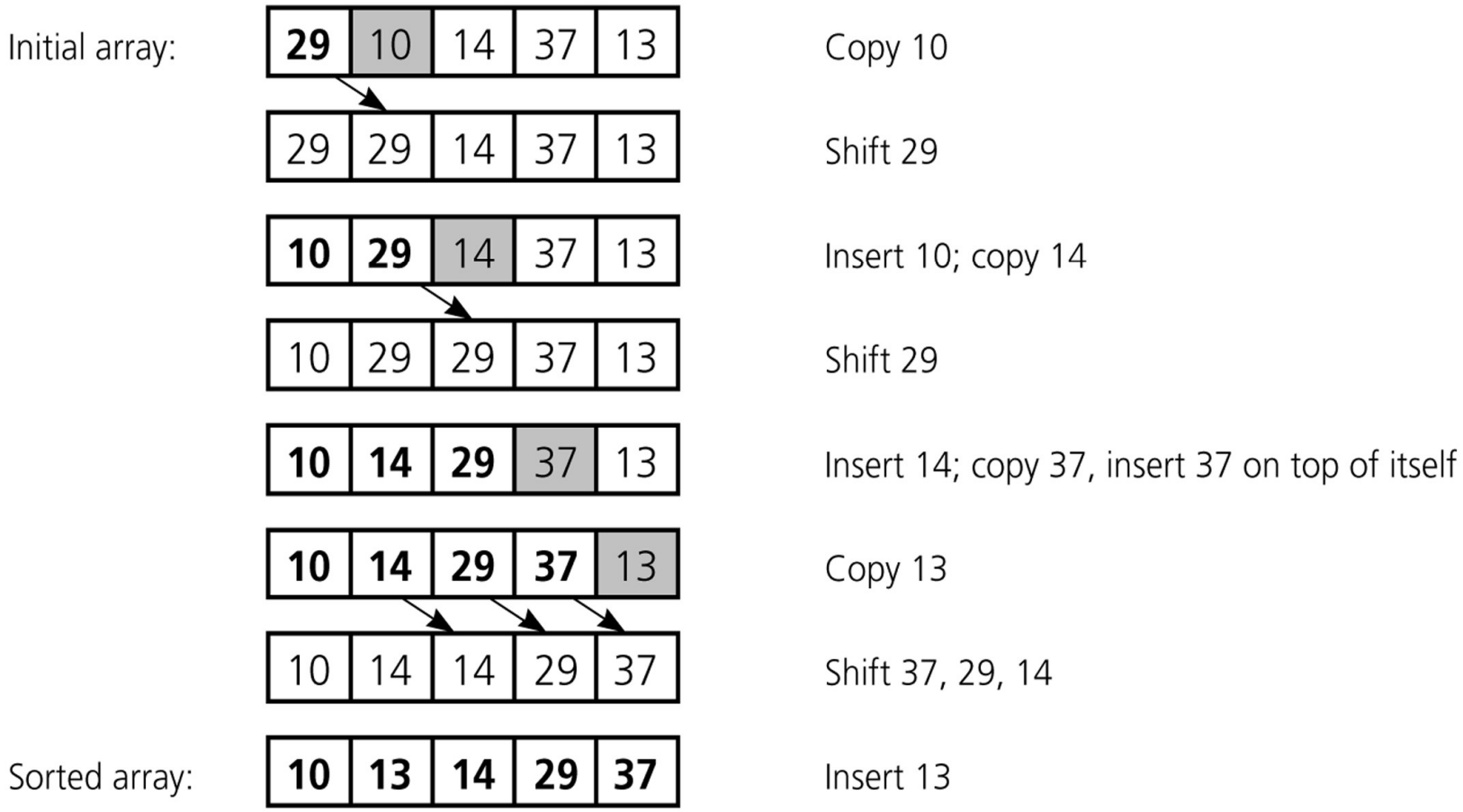
- The loop invariant is the variable k
- Initialization: k = 0 (pre-condition checking) before the iteration and sum = 0 implies that the sum of first zero numbers is 0
- Maintenance: k is incremented by 1 implies that "for a particular k, the value of sum gives the sum of the k numbers
- Termination: When k = n (post-condition checking), the loop terminates and at that time, sum contains the sum of n numbers

Ex. Sorting problem

- Problem Description:
  - Given the sequence of ‘n’ nos a1, a2,…an.
  - Reorder the input sequence such that

    $a1 \leq a2 \leq a3 \leq \ldots. \leq an.$
- Suitable technique: Insertion sort
  - Efficient algorithm for sorting small no. of elements
  - Works the way many people sort hand of playing cards

Insertion Sort Example



Ex. Sorting problem

- Algorithm Design

```
InsertionSort (A[1..n])
    for j = 2 to n
        key = A[j]
        i = j - 1;
        while (i > 0) and (A[i] > key)
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

Proof of Correctness

- We saw how this algorithm works for the given set of elements
- j indicates current card in hand. A[1 … j-1] – sorted cards in hand. A[j+1 …. n] – pile of cards still on the table
- The elements A[1 … j-1] are the elements originally in positions 1 through j-1, but now in sorted order. This property of sub-array A[1 … j-1] is loop invariant.
- At the beginning of each iteration of for loop, the sub-array A[1 … j-1] consists of elements originally in A[1 … j-1], but in sorted order.
- **Loops in an algorithm/program can be proven correct using mathematical induction. In general it involves something called "loop invariant"**

Loop Invariants

- Initialization
  - Check whether loop invariant holds before the first iteration of the for loop (i.e) when j = 2
  - The sub-array A[1 … j-1] has only one element A[1].
  - Sub-array with only one element is by default sorted. So property gets satisfied.

Loop Invariants

- Maintenance
  - Showing that each iteration maintains the loop invariant property
  - It is clear from our example that the property is maintained

Loop Invariants

- Termination
  - Examine whether the loop invariant is maintained when the loop terminates
  - Loop terminates when j = n+1.
  - In this case, sub-array is A[1 … n], which is in sorted order.
  - So property gets satisfied.

Loop Invariants

- Hence we can say that the algorithm is correct.

Measure Time Complexity

- Loops have impact over running time
- 3 cases
  - Best case analysis – usage of algorithm is atleast
  - Worst case analysis – usage of algorithm is atmost
  - Average case analysis – usage of algorithm is average

Various Computing Times

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

n	log$n$	n	$n$log$n$	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,094	262,144	$1.84 * 10^{19}$
128	7	128	896	16,384	2,097,152	$3.40 * 10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15 * 10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34 * 10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79 * 10^{308}$

The Growth Rate of the Six Popular functions

Example

Algorithm: fun(n)

s=0

For i = 1 to n

s = s + i * i

Analysis

- Sum of squares of n non-negative integers
- Loop gets executed for n times
- O(n)

Analysis

- Improvement:
  - (n(n+1)(2n+1))/6

Performance Analysis of an Algorithm

- Analysis of iterative algorithms
- Analysis of recursive algorithms

Analysis of iterative algorithms

```
Main()                          Time        Cost
   {
   int a,b,c,d,i;
   for(i=0;i<=9;i++)            11          C2
         {
          a=a+b;                10          C3
          b=b+1;                10          C4
         }
    d=c+d;                      1           C5
   }
```

$T(n) = C2*11 + C3*10 + C4*10 + C5*1$

$\Rightarrow T(n) = 1*(C5) + 10*(C3+C4) + C2 * 11$

$\Rightarrow T(n) = a$ where a is some constant

$\Rightarrow T(n) =$ Constant time

Analysis of iterative algorithms

	Time	Cost
main()		
{		
int i,j,k,n;		
i=i+1;	1	C2
for(i=1;i<=n;i++)	n+1	C3
{		
j=j+1;	n	C4
}		
k=k+1;	1	C5
}		

$$\Rightarrow T(n) = C2*1 + C3*(n+1) + C4*n + C5*1$$

$$\Rightarrow T(n) = 1*(C2 + C5) + C3*n+C3*1 + C4 *$$

$$\Rightarrow T(n) = 1*(C2+C3+C5) + n*(C3 + C4)$$

$$\Rightarrow T(n) = a + nb \text{ where a and b are constants}$$

$$\Rightarrow T(n) = O(n)$$

Analysis of iterative algorithms

	Time	Cost
main()		
{		
int i,j,a,b;		
for (i=1;i<=n;i++)	n+1	C2
{		
for(j=1;j<=m;j++)	n*(m+1)	C3
{		
a=a+1;	n*m	C4
}		
b=b+1;	n	C5
}		
}		

$\Rightarrow T(n) = C2*(n+1) + C3*(n*(m+1)) + C4*(n*m) + C5*n$

$\Rightarrow T(n) = C2*n + C2*1 + C3*(nm + n) + C4*nm + C5*n$

$\Rightarrow T(n) = 1*(C2) + n*(C2 + C3 + C5) + nm*(C3 + C4)$

$\Rightarrow T(n) = a + n*b + nm*c$ where a, b, c are constatns

$\Rightarrow T(n) = O(nm)$

Analysis of Recursive Algorithms

- Each recursive algorithm has one or more recursive cases and one or more base cases
- Example: Recurrence relation T(n)

$$T(n) = \begin{cases} a & \text{if } n=1 \\ 2T(n/2)+bn+c & \text{if } n>1 \end{cases}$$

- The portion of definition that does not contain T is called base case
- The portion that contains T is called recursive case

Forming Recurrence Relations

```
Void f (int n)
{
    if n > 0 {
        Print n;
        f(n-1); }
}
```

$$T(n) = \begin{cases} a & \text{if } n=0 \\ b+T(n-1) & \text{if } n>0 \end{cases}$$

Reason:

If n = 0, one comparison operation

If n > 0, 2 basic operations (comparison & print), one recursive call with parameter n-1

Exercise Problem - 1

```
Int g (int n)
{
    if(n==1)
        return 2;
    else
        return 3*g(n/2)+g(n/2)+5
}
```

Solution

$$T(n) = \begin{cases} a & \text{if } n=1 \\ b+2T(n/2) & \text{if } n>1 \end{cases}$$

Exercise Problem - 2

```
Fib (int n)
{
    if (n==1 || n==2)
        return 1
    else
        return fib(n-1) + fib(n-2);
}
```

Solution

$$T(n) = \begin{cases} a & \text{if } n=1 \text{ or } n=2 \\ b+T(n-1)+T(n-2) & \text{if } n>2 \end{cases}$$

Methods to solve Recurrence Relations

There are many methods to solve recurrence relations. Few are listed below:

- Master Method
- Recursion Tree Method
- Substitution method

Master Method

- The Master theorem allows us to easily calculate the running time of recursive algorithm in Θ notation
- Not all recurrence relations can be solved with the use of the master theorem
- General syntax of recurrence relation

$$T(n) = a\,T(n/b)+f(n)$$

Case 1

Generic form: $f(n) = O(n^c)$ where $c<\log_b a$

Then, $T(n) = \Theta(n^{\log_b a})$

Example: $T(n) = 8T(n/2) + 1000n^2$

$a = 8, b = 2, f(n) = 1000n^2$

Next, we see if we satisfy the case 1 condition:

$f(n) = O(n^c)$ where $c = 2$

$\text{Log}_b a = \text{Log}_2 8 = 3$. So $c<\log_b a$ is true

Therefore, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

Case 2

Generic form: $f(n) = \Theta (n^c \log^k n)$ where $c=\log_b a$ and $k>=0$

Then, $T(n) = \Theta(n^c \log^{k+1} n)$

Example: $T(n) = 2T(n/2) + 10n$

$a = 2, b = 2, f(n) = 10n$

Next, we see if we satisfy the case 2 condition:

$f(n) = \Theta (n^c \log^k n)$ where $c = 1$ and $k=0$

$\text{Log}_b a = \text{Log}_2 2 = 1$. So $c = \log_b a$ is true

Therefore, $T(n) = \Theta(n^c \log^{k+1} n) = \Theta(n^1 \log^1 n) = \Theta(n \log n)$

Case 3

Generic form: $f(n) = \Omega(n^c)$ where $c > \log_b a$

Then, $T(n) = \Theta(f(n))$

Example: $T(n) = 2T(n/2) + n^2$

$a = 2, b = 2, f(n) = n^2$

Next, we see if we satisfy the case 3 condition:

$f(n) = \Omega(n^c)$ where $c = 2$

$\text{Log}_b a = \text{Log}_2 2 = 1$. So $c > \log_b a$ is true

Therefore, $T(n) = \Theta(f(n)) = \Theta(n^2)$