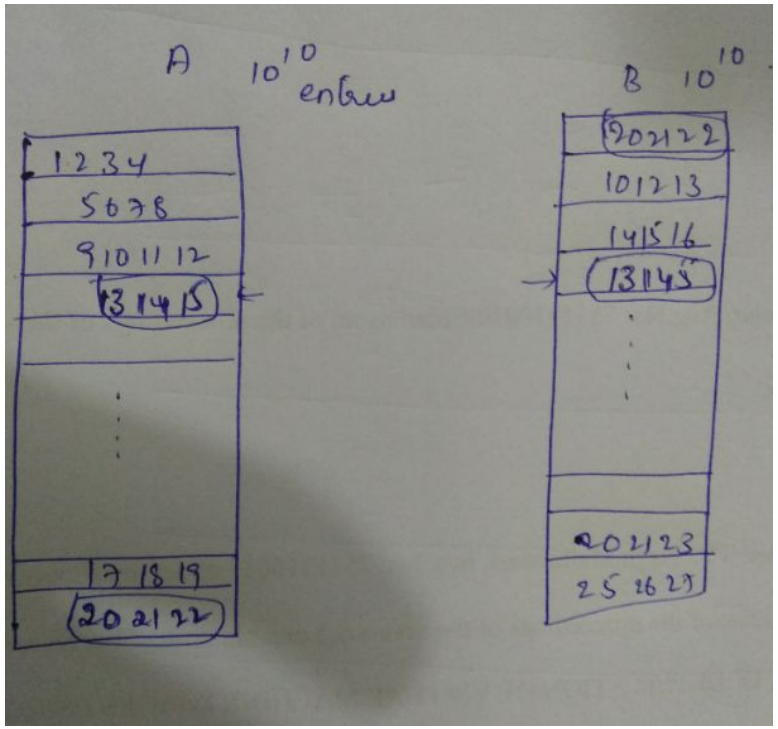
UNIT-I

Introduction-Algorithm Design

- **Session Learning Outcome-SLO**-To understand why algorithm design is important.
- **Example 1 :**

  Take 2 arrays A and B of 10^{10} entries each. Now write an algorithm to find common entries.



Algorithm:

Assume 2 arrays
k=1
for i=1 ton
    for j=1 to n
        if A[i]=B[j] then C[k] = A[i]; k=k+1
Output the array C

- Steps taken by the above algorithm is n^2 that is $(10^{10})^2 = 10^{20}$
- How many seconds 10^{20} steps take?
- 10^{10} Seconds = 317 years
- On a modern powerful computer it takes 3 years.

Motivation:

- Whether any way to write an algorithm for the above problem which takes less time than previously mentioned (Yes/No)

  Yes

Example 2: A book contains page number from 1 and 1024, and I have to search 73 page

Maximum number of comparisons is

512, 256, 128, 64, 32, 16, 8, 4, 2, 1

For the range 1-1024 –10 comparisons

For the range 1-2048 – 11 comparisons

For the range 1-4096- 12 comparisons

For the range 1-N--- ? Comparisons

What is the relation

- It is log N comparisons ($1024 = 2^{10}$), $2048 = 2^{11}$ $4096 = 2^{12}$ …….
- $\log_2 N$ is the number of powers of 2 in N
- $\log_2 N$ is a slow growing function

- Now take example 1, finding common entries . Assume array B is arranged in increasing order ? Can I get a better algorithm. (yes / no)

  Yes

- If the example 2 concept (binary search) is used here. Then algorithm will be

for i=1 to n

look for A[i] in B by doing binary search and

if found store in C

Output the array C

Analysis:

- How many steps it takes. $n(\log_2 n)$
- $n= 10^{10}$
- $n(\log_2 n) = 10^{10} (\log_2 10^{10})$
- 10^{10}steps in 1 second then it for the above steps, it takes $10^{10} (\log_2 10^{10}) / 10^{10} <=40$ secs

Summary:

- More than the power of computers, an improvement in algorithm can make a difference between 317 years and 40 seconds.
- How much time to take for sorting?
- Can I choose $O(n^2)$?
- Quick sort, Mergesort…

Fundamentals of Algorithms

Session Learning Outcome-SLO:

- Able to know what is an algorithm and how to write algorithm / pseudocode.

What is an algorithm?

It is a finite set of instructions or a sequence of computational steps that if followed it accomplishes a task.

Characteristics of an algorithm:

- **Input :** 0 or more input
- **Output:** 1 or more output
- **Definiteness**: clear and unambiguous
- **Finiteness:** termination after finite number of steps
- **Effectiveness:** solved by pen and paper

Describing the Algorithm

The skills required to effectively design and analyze algorithms are entangled with the skills required to effectively describe algorithms. A complete description of any algorithm has four components:

- **What**: A precise specification of the problem that the algorithm solves.
- **How**: A precise description of the algorithm itself.
- **Why:** A proof that the algorithm solves the problem it is supposed to solve.
- **How fast:** An analysis of the running time of the algorithm.

It is not necessary (or even advisable) to develop these four components in this particular order. Problem specifications, algorithm descriptions, correctness proofs, and time analyses usually evolve simultaneously, with the development of each component informing the development of the others.

Specifying the Problem

- To describe an algorithm, the problem is described that the algorithm is supposed to solve.
- Algorithmic problems are often presented using **standard English**, in terms of real-world objects. The algorithm designers, restate these problems in terms of formal, abstract, mathematical objects—numbers, arrays, lists, graphs, trees, and so on
- If the problem statement carries any hidden assumptions, state those assumptions explicitly;
- Specification to be refined as we develop the algorithm. For example, an algorithm may require a particular input representation, or produce a particular output representation, that was left unspecified in the original informal problem description.

- The specification should include just enough detail that someone else could use our algorithm as a black box, without knowing how or why the algorithm actually works.
- In particular, the type and meaning of each input parameter, and exactly how the eventual output depends on the input parameters are described. On the other hand, specification should deliberately hide any details that are not necessary to use the algorithm as a black box.
- **Example:** Given two non-negative integers x and y, each represented as an array of digits, compute the product $x \cdot y$, also represented as an array of digits. To someone using these algorithms, the choice of algorithm is completely irrelevant.

Describing Algorithms

- The clearest way to present an algorithm is using a combination of pseudocode and structured English.
- Pseudocode uses the structure of formal programming languages and mathematics to break algorithms into primitive steps; the primitive steps themselves can be written using mathematical notation, pure English, or an appropriate mixture of the two, whatever is clearest.
- Well written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm easier to understand, analyze, debug, and implement.

Analysing Algorithms:

We have to prove that the algorithm actually does what it's supposed to do, and that it does so efficiently.

Correctness

- In some application settings, it is acceptable for programs to behave correctly most of the time, on all "reasonable" inputs.
- But we require algorithms that are always correct, for all possible inputs.
- We must prove that our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough.
- In particular, correctness proofs usually involve induction.

Analyzing Algorithms:

Running Time

- The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any particular problem.

Pseudocode conventions

1. **Comments** //
2. **Blocks** {and }.
3. An **identifier** begins with a letter
4. ; (dot), -> operator

$$
\begin{array}{l}
node = \textbf{record} \\
\quad \{ \quad datatype\_1 \quad data\_1; \\
\qquad \vdots \\
\quad \quad datatype\_n \quad data\_n; \\
\quad \quad node \qquad\quad *link; \\
\quad \}
\end{array}
$$

1. **Assignment---** $\langle variable \rangle := \langle expression \rangle;$
2. Two **Boolean values *true*** and ***false***, logical operators and, or, and not, relational operators <, ≤, ≥,and >

7. Elements of **multi-dimensional arrays** are accessed using[and]. For example, if A is a two dimensional array, the (i,j)th element of the array is denoted as - A[i, j]. Array indices start at zero.

8. Looping statements- for, while and repeat-until

- while loop written as

```
while ⟨condition⟩ do
{
        ⟨statement 1⟩
              ⋮
        ⟨statement n⟩
}
```

As long as (condition)is true, the statements get executed. When (condition)becomes false, the loop is exited. The value of (condition) is evaluated at the top of the loop.

- The general form of a for loop is

```
for variable := value1 to value2  step step do
{
        ⟨statement 1⟩
                ⋮
        ⟨statement n⟩
}
```

Here valuel, value2, and *step* are arithmetic expression The clause "step *step*" is optional and taken as +1 if it does not occur. *Step* could either be positive or negative

- A repeat-until statement is constructed as follows:

```
repeat
        ⟨statement 1⟩
                ⋮
        ⟨statement n⟩
until ⟨condition⟩
```

9. **break –**
10. **return**
11. **Conditional statement has the following forms**

if ⟨*condition*⟩ **then** ⟨*statement*⟩
if ⟨*condition*⟩ **then** ⟨*statement 1*⟩ **else** ⟨*statement 2*⟩

9. **Multiple-decision statement as:**

case
{
:⟨*condition 1*⟩: ⟨*statement 1*⟩
⋮
:⟨*condition n*⟩: ⟨*statement n*⟩
:else: ⟨*statement n + 1*⟩
}

9. **Input, output :** read, write

14. Procedure: *Algorithm*. An algorithm consists of a heading and a body.

Algorithm *Name* (⟨*parameter list*⟩)

Where *Name* is procedure name and (<parameter list>) is a listing of the procedure parameters. The body has one or more (simple or compound)statements enclosed within braces {and } .

Example of pseudocode:

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

In this algorithm (named Max), A and n are procedure parameters. *Result* and i are local variables.

Summary:

- Every problem is to be specified, for which an algorithm is described in the form of pseudocode and is analysed to test its efficiency.

Home assignment:

1. Write the pseudocode for finding matrix multiplication of 2 two-dimensional arrays.
2. Write the pseudocode for linear search.

Correctness of Algorithm

Session Learning Outcome-SLO:

- Able to prove the correctness of an algorithm by any of the methods.

A proof of correctness requires that the solution be stated in two forms.

- One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus.
- The second form is called a specification, and this may also be expressed in the predicate calculus.

- A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct.
- A proof of correctness is much more valuable than a thousand tests(if that proof is correct),since it guarantees that the program will work correctly for all possible inputs.

Methods of proving correctness

How to prove that an algorithm is correct?

Proof by:

- Counterexample (indirect proof)
- Induction (direct proof)
- Loop Invariant

- **Other approaches:**
  - proof by cases/enumeration
  - proof by chain of iffs
  - proof by contradiction
  - proof by contrapositive

- For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem. For sorting, this means even if the input is already sorted or it contains repeated elements.

Proof by mathematical induction

Mathematical induction (MI) is an essential tool for proving the statement that proves an algorithm's correctness. The general idea of MI is to prove that a statement is true for every natural number n.

It contains 3 steps:

1. **Induction Hypothesis**: Define the rule we want to prove for every n, let's call the rule f(n).
2. **Induction Base**: Proving the rule is valid for an initial value, or rather a starting point - this is often proven by solving the Induction Hypothesis f(n) for n=1 or whatever initial value is appropriate
3. **Induction Step**: Proving that if we know that f(n) is true, we can step one step forward and assume f(n+1) is correct

Example

If we define S(n) as the sum of the first n natural numbers, for example S(3) = 3+2+1, prove that the following formula can be applied to any n:

$$S(n) = \frac{(n+1) * n}{2}$$

Let's trace our steps:

1. **Induction Hypothesis**: S(n) defined with the formula above

1. **Induction Base**: I $S(1) = \frac{(1+1) * 1}{2} = \frac{2}{2} = 1$ 1) = 1

3. **Induction Step**: In this step we need to prove that if the formula applies to S(n), it also applies to S(n+1) as follows:

$$S(n+1) = \frac{(n+1+1)*(n+1)}{2} = \frac{(n+2)*(n+1)}{2}$$

This is known as an **implication** (a=>b), which just means that we have to prove b is correct providing we know a is correct.

$$S(n+1) = S(n) + (n+1) = \frac{(n+1)*n}{2} + (n+1) = \frac{n^2+n+2n+2}{2}$$

$$\boxed{= \frac{n^2+3n+2}{2} = \frac{(n+2)*(n+1)}{2}}$$

Note that S(n+1) = S(n) + (n+1) just means we are recursively calculating the sum.

Example with literals:

S(3) = S(2) + 3= S(1) + 2 + 3 = 1 + 2 + 3 = 6 Hence proved

Summary:

For every algorithm proof of correctness is very important to show that the program is correct in all cases.

Home Assignment

Prove by induction:

(a) $\sum_{i=1}^{n} i = n(n+1)/2,\ n \geq 1$

(b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6,\ n \geq 1$

(c) $\sum_{i=0}^{n} x^i = (x^{n+1}-1)/(x-1),\ x \neq 1,\ n \geq 0$

Time complexity analysis

Session Learning Outcome-SLO:

Able to write algorithms and analyze its complexity

- The time T(P) taken by a program P is

  T(P) =compile time + run (or execution) time.

- The compile time does not depend on the **instance** characteristics.
- Also, the compiled program will be run several times without recompilation. Consequently, the run time of a program only considered. This run time is denoted by t_p (instance characteristics)
- As the factors that influence t_p , when the program run it is better to estimate t_p
- If the compiler characteristics and the time taken for addition, subtraction, division, multiplication etc.. are known then we can find t_p as

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

- Where n denotes the instance characteristics, and c_a,c_s,c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on.

- ADD, SUB,MUL,DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

- It is difficult to get such an exact formula because the time taken for addition and other operations depends on numbers that are being added.

- So alternative is to do all activities such as program typing, compiling and running on a machine and the execution time is physically clocked to get $t_p(n)$.

- Drawbacks of experimental approach is the execution time depends on the other programs running on the computer at the time the program P run. It also depends on machine architecture.

Solution:

- Obtain a count for the total number of operations in the algorithm. We can go one step further and count only the number of **program steps.**

- A **program step** is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement is a single program step.

$$\textbf{return } a + b + b * c + (a + b - c)/(a + b) + 4.0;$$

- The number of steps any program statement is assigned depends on the kind of statement.
- Comments count as zero steps;
- an assignment statement which does not involve any calls to other algorithms is counted as one step;
- In an iterative statement such as the for, while, and repeat-until statements, we consider the step counts only for the control part of the statement.

Two methods to find the time complexity for an algorithm

1. Count variable method
2. Table method

Table method

- The table contains s/e and frequency.
- The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
- Frequency is defined as the total number of times each statement is executed.
- Combining these two, the step count for the entire algorithm is obtained.

Example 1

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n + 3$

Example 2

Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a,n-1)+a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Home assignment

Calculate the time complexity for the below algorithms using table method

1.

```
1   Algorithm Fibonacci(n)
2   // Compute the nth Fibonacci number.
3   {
4       if (n ≤ 1) then
5           write (n);
6       else
7       {
8           fnm2 := 0; fnm1 := 1;
9           for i := 2 to n do
10          {
11              fn := fnm1 + fnm2;
12              fnm2 := fnm1; fnm1 := fn;
13          }
14          write (fn);
15      }
16  }
```

2. Determine the frequency counts for all statements in the following two algorithm segments:

```
1  for i := 1 to n do
2      for j := 1 to i do
3          for k := 1 to j do
4              x := x + 1;
```

(a)

```
1  i := 1;
2  while (i ≤ n) do
3  {
4      x := x + 1;
5      i := i + 1;
6  }
```

(b)

3.

```
1   Algorithm Transpose(a, n)
2   {
3       for i := 1 to n − 1 do
4           for j := i + 1 to n do
5           {
6               t := a[i, j]; a[i, j] := a[j, i]; a[j, i] := t;
7           }
8   }
```

References

1. *Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms, 3rd ed., The MIT Press Cambridge, 2014.*

1. *Ellis Horowitz, Sartajsahni, Sanguthevar, Rajesekaran, Fundamentals of Computer Algorithms, Galgotia Publication, 2010.*

1. *https://www.google.com/search?q=jeff+erickson+algorithms+pdf&oq=jeff&aqs=chrome.2.69i57j46i433j69i59j0i433j46i433l2j46i395i433j0i271.4680j1j7&sourceid=chrome&ie=UTF-8*

Thank you