18MCA34C

DESIGN AND ANALYSIS OF ALGORITHM

UNIT I

Introduction

FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,

Assistant Professor,

Postgraduate Department of Computer Applications,

Government Arts College (Autonomous),

Coimbatore-641018.

18MCA34C DESIGN AND ANALYSIS OF ALGORITHM

UNIT I: Algorithm Specification -Recursive Algorithms - Performance Analysis - Space Complexity - Time Complexity -Asymptotic Notations - Asymptotic Complexity of SUM and Recursive SUM and ADD Algorithms - Analysis of Sequential Search.

UNIT II: Elementary Data Structures- Stacks and Queues - Trees - Binary Trees - Binary Search Trees - Iterative and Recursive Search of BST - Graphs - Konigsberg Bridge Problem - Graph Representations - Graph Traversals.

UNIT III: Divide and Conquer: General Method - Binary Search - Finding Maximum and Minimum - Merge Sort - Greedy Algorithms: General Method - Container Loading - Knapsack Problem.

UNIT IV: Dynamic Programming: General Method - Multistage Graphs - All-Pair shortest paths - Optimal binary search trees - 0/1 Knapsack - Travelling salesperson problem.

UNIT V: Backtracking: General Method - 8 Queens Problem - sum of subsets - graph coloring - Hamiltonian problem - knapsack problem.

TEXT BOOKS:

1. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Fundamentals of Computer Algorithm, Galgotia Publications, 2007.

REFERENCE BOOKS:

1. T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2003.

2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, "The Design and analysis of Computer Algorithms", Pearson Education, 1999.

WHAT IS AN ALGORITHM?

An algorithm is a finite set of instructions that,if followed, accomplishes a particular task

All algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

There are four distinct areas of algorithm study

1. How to devise algorithms ?
2. How to validate algorithm?
3. How to analyze algorithm?

   Analysis of algorithms or performance analysis refers to the task of determining how much

   Computing time and storage an algorithm requires.
4. How to test a program Testing a program?

   It consists of two phases:

   Debugging and profiling (or performance measurement).

   Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, ifso,to correct them

   Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results

ALGORITHM SPECIFICATION

1.2.1 Pseudocode Conventions

1.Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces: { and }.

A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Assignment of values to variables is done using the assignment statement

(variable) := (expression);

5.Thereare two boolean values true and false. In orderto produce these values, the logical operators and, or, and notand the relational operators <,<,=,/, >, and > are provided.

6. Elements of multidimensional 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

7. The following looping statements are employed: **for**, **while**, and **repeat-until**. The **while** loop takes the following form:

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* := *value*1 **to** *value*2 **step** *step* **do**
{
    ⟨*statement* 1⟩
        ⋮
    ⟨*statement n*⟩
}

Here *value*1, *value*2, and *step* are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "**step** *step*" is optional and taken as +1 if it does not occur. *step* could either be positive or negative. *variable* is tested for termination at the start of each iteration. The **for** loop can be implemented as a **while** loop as follows:

variable := *value*1;
fin := *value*2;
incr := *step*;
while $((variable - fin) * step \leq 0)$ **do**
{
    ⟨*statement* 1⟩
        ⋮
    ⟨*statement n*⟩
    variable := *variable* + *incr*;
}

A **repeat-until** statement is constructed as follows:

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

The statements are executed as long as ⟨*condition*⟩ is **false**. The value of ⟨*condition*⟩ is computed after executing the statements.

8. A conditional statement has the following forms:

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

Here ⟨*condition*⟩ is a boolean expression and ⟨*statement*⟩, ⟨*statement* 1⟩, and ⟨*statement* 2⟩ are arbitrary statements (simple or compound).

We also employ the following **case** statement:

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

Here (statement 1), (statement 2), etc. could be either simple statements or compound statements.

A case statement is interpreted as follows.

If (condition 1) is true, (statement1) gets executed and the case statement is exited.

If statement 1) is false, (condition 2) is evaluated. If (condition 2) is true, (statement 2) gets executed and the case statement exited, and soon.

If none of the conditions (condition 1), ... , (condition n) are true, (statement n+1) is executed

and the case statement is exited. The else clause is optional.

9. Input and output are done using the instructions read and write.No format is used to Specify the size of input or output quantities.

10. There is only one type of procedure: Algorithm.

An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name ((parameter list))

Where Name is the name of the procedure and

({parameter list)) is a listing of the procedure parameters.

The body has one or more (simple or compound) statements enclosed within braces { and }.

An algorithm may or may not return any values.

Simple variablesto procedures are passed by value.

Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type.

As an example, the following algorithm finds and returns the maximum of n given numbers:

```
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.

Example 1.1 [Selection sort] Suppose we must devise an algorithm that sorts a collection of n > 1 elements of arbitrary type.

A simple solution is given by the following

From those elements that are currently unsorted, find the smallest and place

it next in the sorted list.

We assume that the elements are stored in an array a, such that the ith integer is stored in the ith position a[i], 1< i < n.

Algorithm 1.1 is our first attempt at deriving a solution

```
1   for i := 1 to n do
2   {
3       Examine a[i] to a[n] and suppose
4       the smallest element is at a[j];
5       Interchange a[i] and a[j];
6   }
```

Algorithm 1.1 Selection sort algorithm

To turn Algorithm 1.1 into a pseudocode program, two clearly defined subtasks remain: finding the smallest element (say $a[j]$) and interchanging it with $a[i]$. We can solve the latter problem using the code

$$t := a[i];\ a[i] := a[j];\ a[j] := t;$$

```
1   Algorithm SelectionSort(a, n)
2   // Sort the array a[1 : n] into nondecreasing order.
3   {
4       for i := 1 to n do
5       {
6           j := i;
7           for k := i + 1 to n do
8               if (a[k] < a[j]) then j := k;
9           t := a[i]; a[i] := a[j]; a[j] := t;
10      }
11  }
```

Algorithm 1.2 Selection sort

1.2.2 Recursive Algorithms

- A recursive function is a function that is defined in terms of itself. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is direct recursive. Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

Factorial fits this category, as well as binomial coefficients, where

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}$$

The *pigeonhole principle* states that if a function f has n distinct inputs but less than n distinct outputs, then there exist two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Present an algorithm to find a and b such that $f(a) = f(b)$. Assume that the function inputs are $1, 2, \ldots,$ and n.

If S is a set of n elements, the *powerset* of S is the set of all possible subsets of S. For example, if $S = (a, b, c)$, then $powerset(S) = \{(\ ), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)\}$. Write a recursive algorithm

1.3 PERFORMANCE ANALYSIS

There are many criteria upon which we can judge an algorithm.

For instance:

1.Does it do what we want it to do?

2. Does it work correctly according to the original specifications of the task?

3. Isthere documentation that describes how to use it and how it works?

4. Are procedures created in such a way that they perform logical sub-functions ?

5. Is the code readable?

Space/Time complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Performance evaluation can be loosely divided into two major phases:

(1)aprioriestimates (performance analysis) and

(2) a posteriori testing(performance measurement)

1.3.1 Space Complexity

Algorithm abc (Algorithm 1.5) computes $a+b+b*c+(a+b-c)/(a+b)+4.0$; Algorithm Sum (Algorithm 1.6) computes $\sum_{i=1}^{n} a[i]$ iteratively, where the $a[i]$'s are real numbers; and RSum (Algorithm 1.7) is a recursive algorithm that computes $\sum_{i=1}^{n} a[i]$.

```
1   Algorithm abc(a, b, c)
2   {
3       return  a + b + b * c + (a + b − c)/(a + b) + 4.0;
4   }
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

The space needed by each of these algorithms is seen to be the sum of the following component

```
1   Algorithm Sum(a, n)
2   {
3       s := 0.0;
4       for i := 1 to n do
5           s := s + a[i];
6       return s;
7   }
```

Algorithm 1.6 Iterative function for sum

```
1   Algorithm RSum(a, n)
2   {
3       if (n ≤ 0) then return 0.0;
4       else return RSum(a, n − 1) + a[n];
5   }
```

Algorithm 1.7 Recursive function for sum

```
Algorithm Add(a,b, c,m,n)
{
        for i := 1to m do
         for
        j := 1to n do
        c[ij] :=a[i,j] +b[i,j];

}
```

Algorithm 1.11 Matrix addition

- A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs.
- This part typically includes the instruction space(i.e., space for the code), space for simple variables and fixed-size component variables (also called aggregate), space for constants,
- and so on.
- A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved,
- the space needed by referenced variables and the recursion stack space (this space depends on the instance characteristics)
- The space requirement S(P) of any algorithm P may therefore be written as $S(P) = c + S_p$(instance characteristics),

Where c is a constant

- When analyzing the space complexity of an algorithm, we concentrate solely on
  estimating S_p (instance characteristics).
- For any given problem, determine which instance characteristics to use to measure the space requirements.
- This is very problem specific, and we resort to examples to illustrate the various possibilities.
- Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the algorithm.
- At times, more complex measures of the interrelationships among the data items are used.

1.3.2 Time Complexity

- The time T(P) taken by a program P is the sum of the compile time and the run(or execution) time.
- The compile time does not depend on the instance characteristics.
- This run time is denoted by tp (instance characteristics)
- Because many of the factors tp depends on are not known at the time a program is conceived, it is reasonable to attempt only to estimate tp.
- If we knew the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P.

So, we could obtain an expression for $t_P(n)$ of the form

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

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, and
- ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions that are performed when the code for P is used on an instance with characteristic n.
- since the time needed for an addition, subtraction, multiplication, and so on, often depends on the numbers being added, subtracted, multiplied, and so on.
- The value of tp(n) for any given n can be obtained only experimentally.
- The program is typed, compiled, and run on a particular machine.
- The execution time is physically clocked, and tp(n) obtained
- 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

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

of Algorithm 1.5 could be regarded as a step since its execution time is independent of the instance characteristics.

- The number of steps any program statement is assigned depends on the kind of statement.
- For example, 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

Asymptotic Notation (O, Ω, Θ)

Definition 1.4 [Big "oh"] The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$. □

Definition 1.5 [Omega] The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$. □

Definition 1.6 [Theta] The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$. □

Asymptotic Complexity of SUM and Recursive SUM and ADD Algorithms

- The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.
- This figure is often arrived at by first determining the number of steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed.
- The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
- By combining these two quantities, the total contribution of each statement is obtained.
- By adding the contributions of all statements, the step count for the entire algorithm is obtained.
- In Table 1.1, the number of steps per execution and the frequency of each of the statements in Sum (Algorithm 1.6) have been listed.
- The total number of steps required by the algorithm is determined to be 2n + 3.
- It is important to note that the frequency of the for statement is n + 1 and not n.
- This is so because i has to be incremented to n + 1 before the for loop can terminate.

- Table1.2 gives the step count for RSum (Algorithm 1.7). Notice that under the s/e(steps per execution) column, the else clause has been given a count of $1+ t_{RSum}(n-1)$.
- This is the total cost of this line each time it is executed.
- It includes all the steps that get executed as a result of the invocation of RSum from the else clause.

- The frequency and total steps columns have been split into two parts:
- one for the case n = 0 and the other for the case n > 0
- Table1.3 corresponds to algorithm Add (Algorithm 1.11).
- Once again, note that the frequency of the first for loop is m + 1 and not m.
- This is so as i needs to be incremented up to m + 1before the loop can terminate.
- Similarly, the frequency for the second for loop is m(n + 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				$2n + 3$

Table 1.1 Step table for Algorithm 1.6

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	$2+x$

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

Table 1.2 Step table for Algorithm 1.7

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	mn	mn
6 }	0	–	0
Total			$2mn+2m+1$

Table 1.3 Step table for Algorithm 1.11

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

Table 1.4 Asymptotic complexity of Sum (Algorithm 1.6)

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

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

Table 1.5 Asymptotic complexity of RSum (Algorithm 1.7).

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 **for** $i := 1$ **to** m **do**	1	$\Theta(m)$	$\Theta(m)$
4 **for** $i := 1$ **to** n **do**	1	$\Theta(mn)$	$\Theta(mn)$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	$\Theta(mn)$	$\Theta(mn)$
6 }	0	–	$\Theta(0)$
Total			$\Theta(mn)$

Table 1.6 Asymptotic complexity of Add (Algorithm 1.11)

Performance Measurement

- Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm
- Suppose we wish to measure the worst-case performance of the sequential search algorithm (Algorithm 1.17).
- Before we can do this, we need to decide on

  1)the values of n for which the times are to be obtained and

  2)determine, for each of the above values of n, the data that exhibit the worst-case behavior

Analysis of sequential search

```
1   Algorithm SeqSearch(a, x, n)
2   // Search for x in a[1 : n]. a[0] is used as additional space.
3   {
4       i := n; a[0] := x;
5       while (a[i] ≠ x) do i := i − 1;
6       return i;
7   }
```

Algorithm 1.17 Sequential search

- The decision on which values of n to use is based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained
- Asymptotic analysis tells us the behavior only for sufficiently large values of n.
- For smaller values of n, the run time may not follow the asymptotic curve.

THANK YOU

This content is taken from the text books and reference books prescribed in the syllabus.