**Algorithms**

### An algorithm is a type of effective method in which a definite list of well-defined instructions for completing a task; that given an initial state, will proceed through a well-defined series of successive states, eventually terminating in an end-state. The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as finding the common divisor of two numbers or multiplying two numbers.

### Algorithms are named for the 9th century Persian mathematician Al-Khowarizmi. He wrote a treatise in Arabic in 825 AD, *On Calculation with Hindu Numerals.* It was translated into Latin in the 12th century as *Algoritmi de numero Indorum*, which title was likely intended to mean "[Book by] Algoritmus on the numbers of the Indians", where "Algoritmi" was the translator's rendition of the author's name in the genitive case; but people misunderstanding the title treated *Algoritmi* as a Latin plural and this led to the word "algorithm" (Latin *algorismus*) coming to mean "calculation method".

**Algorithm Specification**

### The criteria for any set of instruction for an algorithm is as follows:

* Input : Zero of more quantities that are externally applied

### Output : At least one quantity is produced

### Definiteness : Each instruction should be clear and unambiguous

### Finiteness : Algorithm terminates after finite number of steps for all test cases.

### Effectiveness : Each instruction is basic enough for a person to carried out using a pen and paper. That means ensure not only definite but also check whether feasible or not.

**Algorithm Classification**

There are various ways to classify algorithms. They are as follows

**Classification by Design Paradigm**

**Divide and conquer**. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively), until the instances are small enough to solve easily. One such example of divide and conquer is [merge sorting](file:///E:\wiki\Mergesort). Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in conquer phase by merging them. A simpler variant of divide and conquer is called decrease and conquer algorithm, that solves an identical sub problem and uses the solution of this sub problem to solve the bigger problem. Divide and conquer divides the problem into multiple sub problems and so conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is binary search algorithm.

**Dynamic programming**. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to sub problems, and overlapping sub problems, meaning the same sub problems are used to solve many different problem instances, a quicker approach called *dynamic programming* avoids recomputing solutions that have already been computed. For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that sub problems are more or less independent in divide and conquer, whereas sub problems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When sub problems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex problems. By using memoization or maintaining a table of sub problems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

**The greedy method**. A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the sub problems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. The greedy method extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best decision (not all possible decisions) made in previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Kruskal.

**Linear programming**. When solving a problem using linear programming, specific inequalities involving the inputs are found and then an attempt is made to maximize (or minimize) some linear function of the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm.

**Areas of research under study of Algorithms**

* How to devise Algorithm :-

Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to devise new and useful algorithms

* How to validate Algorithm-

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs independent of issue concerning the programming language it eventually be written in.

* How to Analyze Algorithms

Analysis of Algorithm or performance analysis refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area which some times requires great mathematical skill

* How to test a program

Testing a programs consist of two phases

* Debugging:- Debugging is the process of executing the programs on sample data sets to determine whether faulty result occur and if so , to correct them.
* Profiling or performance measurement is the process of executing a correct program on data set and measuring the time and space it take to compute the result.

**Algorithm Specification**

We can use natural languages like English.Graphic representation is called flow chart.Another representation is Pseudocode that resembles C or Pascal

1. Comments begin with //
2. Blocks are indicated with matching braces { }
3. An identifier begins with a letter
4. Assignment of values to variables is done using the assignment statement

< variable> :=<expression>

5. Logical operators and or not and the relational operators <, ≤, =,≠,>, ≥ are provided

6. The following looping statements are employed:

for, while, repeat-until

7. The following conditional statements can be used

If……. Then….

if …….then……else……

**Switch** (<expr>) {

**Case** cond1 : <statement1>

**Case** cond2 : <statement2>

.

.

**Default** : <statement>

}

8. Inputs and outputs are done using the instructions read and write

9. There is only one procedure: Algorithm

the heading takes the form

**Algorithm** Name ( parameter list)

**Performance Analysis**

To improve existing algorithms .

Two types

– Apriori Analysis

– Aposteriori Analysis

**Apriori Analysis - Example**

Doing an analysis of the solutions before performing the action. Physical Example- Find path from P to Q , Criteria for selection – Path length, road conditions, type of vehicle, speed

Doing an analysis of the solutions before coding the algorithms. Given two or more algorithms for a problem – Doing a machine independent analysis to find better algorithm.

**Space Complexity**

Space complexity of an algorithm is the amount to memory needed by the program for its completion. Space needed by a program has the following components:

1. **Instruction Space**

Space needed to store the compiled version of program. It depends on

* 1. Compiler used
  2. Options specified at the time of compilation

e.g., whether optimization specified, Is there any overlay option etc.

* 1. Target computer

e.g., For performing floating point arithmetic, if hardware present or not.

1. **Data Space**

Space needed to store constant and variable values. It has two components:

* 1. Space for constants:

e.g., value ‘3’ in program 1.1

Space for simple variables:

e.g., variables a,b,c in program 1.1

**Program 1.1**

int add (int a, int b, int c)

{

return (a+b+c)/3;

}

* 1. Space for component variables like arrays, structures, dynamically allocated memory.

e.g., variables a in program 1.2

**Program 1.2**

int Radd (int a[], int n)

1 {

2 If (n>0)

3 return Radd (a, n-1) + a[n-1];

4 else

5 return 0;

6 }

1. **Environment stack space**

Environment stack is used to store information to resume execution of partially completed functions. When a function is invoked, following data are stored in Environment stack.

* 1. Return address.
  2. Value of local and formal variables.
  3. Binding of all reference and constant reference parameters.

Space needed by the program can be divided into two parts.

i. Fixed part independent of instance characteristics. E.g., code space, simple variables, fixed size component variables etc.

1. Variable part. Space for component variables with space depends on particular instance. Value of local and formal variables.

Hence we can write the space complexity as

S(P) = c + Sp (instance characteristics)

**Example 1.1**

Refer Program 1.1

One word for variables a,b,c. No instance characteristics. Hence Sp(TC) = 0

**Example 1.2**

**Program 1.3**

int Aadd (int \*a, int n)

1 {

2 int s=0;

3 for (i=0; i<n; i++)

4 s+ = a[i];

5 return s;

6 }

One word for variables n and i. Space for a[] is address of a[0]. Hence it requires one word. No instance characteristics. Hence Sp(TC) = 0

**Example 1.3**

Refer Program 1.2

Instance characteristics depend on values of n. Recursive stack space includes space for formal parameters, local variables and return address. So one word each for a[],n, return address and return variables. Hence for each pass it needs 4 words. Total recursive stack space needed is 4(n).

Hence Sp(TC) = 4(n).

**Time Complexity**

Time complexity of an algorithm is the amount of time needed by the program for its completion. Time taken is the sum of the compile time and the execution time. Compile time does not depend on instantaneous characteristics. Hence we can ignore it.

Program step: A program step is syntactically or semantically meaningful segment of a program whose execution time is independent of instantaneous characteristics. We can calculate complexity in terms of

1. Comments:

No executables, hence step count = 0

1. Declarative Statements:

Define or characterize variables and constants like (*int* , *long*, *enum*, …)

Statement enabling data types (*class, struct, union, template*)

Determine access statements ( *public, private, protected, friend* )

Character functions ( *void, virtual* )

All the above are non executables, hence step count = 0

1. Expressions and Assignment Statements:

Simple expressions : Step count = 1. But if expressions contain function call, step count is the cost of the invoking functions. This will be large if parameters are passed as call by value, because value of the actual parameters must assigned to formal parameters.

Assignment statements : General form is <variable> = <expr>. Step count = expr, unless size of <variable> is a function of instance characteristics. eg., a = b, where a and b are structures. In that case, Step count = size of <variable> + size of < expr >

1. Iterative Statements:

**While** <expr> **do**

**Do .. While <**expr**>**

Step count = Number of step count assignable to <expr>

**For** (<init-stmt>; <expr1>; <expr2>)

Step count = 1, unless the <init-stmt>, <expr1>,<expr2> are function of instance characteristics. If so, first execution of control part has step count as sum of count of <init-stmt> and <expr1>. For remaining executions, control part has step count as sum of count of <expr1> and <expr2>.

1. Switch Statements:

**Switch** (<expr>) {

**Case** cond1 : <statement1>

**Case** cond2 : <statement2>

.

.

**Default** : <statement>

}

**Switch** (<expr>) has step count = cost of <expr>

Cost of **Cond** statements is its cost plus cost of all preceding statements.

1. If-else Statements:

**If** (<expr>) <statement1>;

**Else** <statement2>;

Step count of **If** and **Else** is the cost of <expr>.

1. Function invocation:

All function invocation has Step count = 1, unless it has parameters passed as called by value which depend s on instance characteristics. If so, Step count is the sum of the size of these values.

If function being invoked is recursive, consider the local variables also.

1. Memory management Statements:

**new** object, **delete** object, **sizeof**(object), Step count =1.

1. Function Statements:

Step count = 0, cost is already assigned to invoking statements.

1. Jump Statements:

**continue, break, goto** has Step count =1

**return** <expr>: Step count =1, if no *expr* which is a function of instance characteristics. If there is, consider its cost also.

**Example 1.4**

Refer Program 1.2

Introducing a counter for each executable line we can rewrite the program as

int Radd (int a[], int n)

{

count++ // if

If (n>0)

{

count++ // return

return Radd (a, n-1) + a[n-1];

}

else

{

count++ // return

return 0;

}

}

Case 1: n=0

tRadd = 2

Case 2: n>0

2 + tRadd (n-1)

= 2 + 2 + tRadd (n-2)

= 2 \* 2 + tRadd (n-2)

.

.

= 2n + tRadd (0)

= 2n + 2

**Example 1.5**

**Program 1.4**

int Madd (int a[][], int b[][], int c[][], int n)

1 {

2 For (int i=0; i<m; i++)

3 For (int j=0; j<n; j++)

4 c[i][j] = a[i][j] + b[i][j];

5 }

Introducing a counter for each executable line we can rewrite the program as

int Madd (int a[][], int b[][], int c[][], int n)

{

For (int i=0; i<m; i++)

{

count++ //for i

For (int j=0; j<n; j++)

{

count++ //for j

c[i][j] = a[i][j] + b[i][j];

count++ //for assignment

}

count++ //for last j

}

count++ //for last i

}

Step count is 2mn + 2m +1.

Step count does not reflect the complexity of statement. It is reflected in step per execution (s/e).

Refer Program 1.2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Line** | **s/e** | **Frequency** | | **Total Steps** | |
| **n=0** | **n>0** | **n=0** | **n>0** |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 |
| 3 | 1 + tRadd (n-1) | 0 | 1 | 0 | 1 + tRadd (n-1) |
| 4 | 0 | 1 | 0 | 0 | 0 |
| 5 | 1 | 1 | 0 | 1 | 0 |
| **Total no. of steps** | | | | **2** | **2 + tRadd (n-1)** |

Refer Program 1.3

|  |  |  |  |
| --- | --- | --- | --- |
| **Line** | **s/e** | **Frequency** | **Total Steps** |
| 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | n+1 | n+1 |
| 4 | 1 | n | N |
| 5 | 1 | 1 | 1 |
| 6 | 0 | 1 | 0 |
| **Total no. of steps** | | | **2n + 3** |

Refer Program 1.4

|  |  |  |  |
| --- | --- | --- | --- |
| **Line** | **s/e** | **Frequency** | **Total Steps** |
| 1 | 0 | 1 | 0 |
| 2 | 1 | m+1 | m+1 |
| 3 | 1 | m(n+1) | m(n+1) |
| 4 | 1 | mn | Mn |
| 5 | 0 | 1 | 0 |
| **Total no. of steps** | | | **2mn + 2m + 1** |

* 1. **Asymptotic Notations**

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Since the values are not exact quantities we need only comparative statements like c1n2 ≤ tp(n) ≤ c2n2.

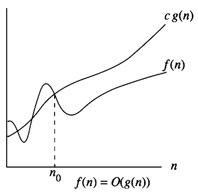
For example, consider two programs with complexities c1n2 + c2n and c3n respectively. For small values of n, complexity depend upon values of c1, c2 and c3. But there will also be an n beyond which complexity of c3n is better than that of c1n2 + c2n.This value of n is called break-even point. If this point is zero, c3n is always faster (or at least as fast). Common asymptotic functions are given below.

|  |  |
| --- | --- |
| **Function** | **Name** |
| 1 | Constant |
| log n | Logarithmic |
| N | Linear |
| n log n | n log n |
| n2 | Quadratic |
| n3 | Cubic |
| 2n | Exponential |
| n! | Factorial |

**1.5.1 Big ‘Oh’ Notation (O)**

O(g(n)) = { f(n) : there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as

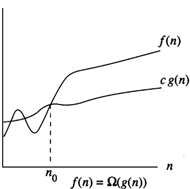
****

**Fig 1.1**

**Omega Notation (Ω)**

Ω (g(n)) = { f(n) : there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as

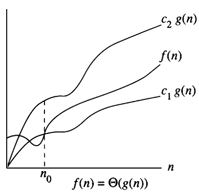


**Fig 1.2**

**Theta Notation (Θ)**

Θ(g(n)) = {f(n) : there exist positive constants c1,c2 and n0 such that c1g(n) ≤f(n) ≤c2g(n) for all n ≥ n0 }

If f(n) = Θ(g(n)), all values of n right to n0 f(n) lies on or above c1g(n) and on or below c2g(n). Hence it is asymptotic tight bound for f(n).



**Fig 1.3**

**Little ‘Oh’ Notation (o)**

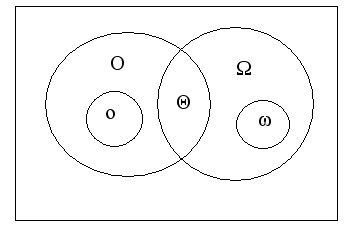
o(g(n)) = { f(n) : for any positive constants c > 0, there exists n0>0, such that 0 ≤ f(n) < cg(n) for all n ≥ n0 }

It defines the asymptotic *tight* upper bound. Main difference with Big Oh is that Big Oh defines for some constants c by Little Oh defines for all constants.

**Little Omega (ω)**

ω(g(n)) = { f(n) : for any positive constants c>0 and n0>0 such that 0 ≤ cg(n) < f(n) for all n ≥ n0 }

It defines the asymptotic *tight* lower bound. Main difference with Ω is that, ω defines for some constants c by ω defines for all constants.

* 1. ****  
     **Recurrence Relations**

Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases

e.g., recurrence for Merge-Sort



* Useful for analyzing recurrent algorithms
* Make it easier to compare the complexity of two algorithms
* Methods for solving recurrences
  + Substitution method
  + Recursion tree method
  + Master method
  + Iteration method
    1. **Substitution Method**
    - Use mathematical induction to derive an answer
    - Derive a function of n (or other variables used to express the size of the problem) that is not a recurrence so we can establish an upper and/or lower bound on the recurrence
    - May get an exact solution or may just get upper or lower bounds on the solution
    1. **Recursion tree Method**
* Main disadvantage of Substitution method is that it is always difficult to come up with a good guess
* Recursion tree method allows you make a good guess for the substitution method
* Allows to visualize the process of iterating the recurrence

**Steps**

* Convert the recurrence into a tree.
* Each node represents the cost of a single sub problem somewhere in the set of recursive function invocations
* Sum the costs within each level of the tree to obtain a set of per-level costs
* Sum all the per-level costs to determine the total cost of all levels of the recursion
  + 1. **Master Method**
* The master method applies to recurrences of the form *T*(*n*) = *a T*(*n*/*b*) + *f* (*n*) ,

where *a* ≥ 1, *b* > 1, and *f* is asymptotically positive.

* Describe the running time of an algorithm that divides a problem of size *n* into *a* sub problems, each of size *n/b*
* There are three common cases

**Case 1**

* Compare *f* (*n*) with *n*log*ba*
* If *f* (*n*) = *O*(*n*log*ba* – e) for some constant e > 0.

i.e., *f* (*n*) grows polynomially slower than *n*log*ba*

i.e., *f(n)* is asymptotically smaller by an *n*e factor.

Then ***Solution:*** *T*(*n*) = Θ(*n*log*ba*) .

**Case 2**

* Compare *f* (*n*) with *n*log*ba*
* If *f* (*n*) = Θ (*n*log*ba*)

i.e., *f* (*n*) and *n*log*ba* grow at similar rates.

Then ***Solution:*** *T*(*n*) = Θ(*n*log*ba* lg*n*) .

**Case 3**

* Compare *f* (*n*) with *n*log*ba*
* If *f* (*n*) = Ω(*n*log*ba+e*) for some constant e > 0

i.e., *f* (*n*) grows polynomially faster than *n*log*ba*

i.e., *f(n)* is asymptotically larger by an *n*e factor.

Then ***Solution:*** *T*(*n*) = Θ(*f(n)*) .

**Example 1.22**

T(n)=9T(n/3) + n

a=9, b=3, f(n) = n

CASE 1:

T(n) = θ(n2)

**Example 1.23**

*T*(*n*) = 4*T*(*n*/2) + *n*

*a =* 4, *b* = 2 ⇒ *n*log*ba* = *n*2; *f* (*n*) = *n.*

CASE 1: *f* (*n*) = *O*(*n*2– e) for e = 1.

∴ *T*(*n*) = Θ(*n*2).

**Example 1.24**

T(n) = T(2n/3) + 1

a=1, b=3/2, f(n)=1

Case 2:

T(n) = θ(lg *n*)

**Example 1.25**

*T*(*n*) = 4*T*(*n*/2) + *n*2

*a =* 4, *b* = 2 ⇒ *n*log*ba* = *n*2; *f* (*n*) = *n*2*.*

CASE 2: *f* (*n*) = Q(*n*2lg0*n*), that is, *k* = 0.

∴ *T*(*n*) = Θ(*n*2lg *n*).

**Example 1.26**

T(n) = 3T(n/4) + n lg n (Case 3)

a=3, b=4, f(n) = n lg n

 Case3: e = 0.2

For sufficiently large n, af(n/b) = 3(n/4)lg(n/4) ≤ (3/4)n lg n=cf(n) for c=3/4

T(n) = θ(n lg n)

**Example 1.27**

*T*(*n*) = 4*T*(*n*/2) + *n*3

*a =* 4, *b* = 2 ⇒ *n*log*ba* = *n*2; *f* (*n*) = *n*3*.*

CASE 3: *f* (*n*) = Ω(*n*2+ e) for e = 1 *and* 4(*cn*/2)3 £ *cn*3 (reg. cond.) for *c* = 1/2.

∴ *T*(*n*) = Θ(*n*3).