Fundamentals of Algorithmic Problem Solving

1. Understanding the problem

The problem given should be understood completely. Check if it is similar to some standard problems algorithm exists.

Otherwise, a new algorithm has to be devised.

Creating an algorithm is an art which may never by fully automated.

2. Ascertain the capabilities of the computational device

Once the problem is understood we need to know the capabilities of the computing device. This can be done by knowing the type of the architecture, speed and memory availability.

3. Exact or Approximate Solution

Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs. The solution is stated in two forms exact or approximate solution.

4. Decide on the appropriate data structure

Some algorithms do not demand any ingenuity in representing their inputs. Some others are infact are predicted on ingenious data structures.

A data-type is a well-defined collection of data with well-defined set of operations on it. A data structure is an actual implement-ation of a particular abstract data type.

5. Algorithm design techniques

Creating an algorithm is an art which may never be fully auto-mated. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Dynamic programming is one such technique.

6. Methods of specifying an algorithm

Two options for specifying an algorithm. A pseudocode is a mix-ture of natural language and programming language like constructs. A flow chart is a method of expressing an algorithm by a collection of connected geometric shapes.

7. Proving an algorithm correctness

Once algorithm is derised, it is necessary to show that it computes answer for all the possible legal inputs. The process of validation -nt of issues concerning programming languages.

A proof of correctness :

a. Program which is annotated by a set of assertions about the input and output variables of a program.
b. A specification and this may be expressed in predicate calculus.

8. Analyzing algorithms

An algorithm is executed, it uses the computers CPU to perform operations and its memory to hold the program and data. Analysis of algorithms and performance analysis refers to the task of determining how much computing time and storage is required.

9. Coding an algorithm

Programming an algorithm presents both a peril and an oppurtuni -ty. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

A working program prorides an additional oppurtunity in allow-ing an empirical analysis of the underlying algorithm. The analysis is based on timing the program on inputs and the results obtained.

Analysis Framework

A general framework for analyzing the efficiency of algorithms is categorized into two types

a. Time efficiency : indicates how fast an algorithm in question runs.
b. Space efficiency : deals with the extra space of the algorithm requires.

The analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them. This can be measured by :

a. Measuring an input's size.
b. Measuring running time.
c. Orders of growth based on input size.

Performance Analysis

Space Complexity

The space complexity of an algorithm is the amount of memory required to run the program completely and efficiently.

The effficiency is measured with respect to the space (memory management), the word space complexity is often used.

$$S(P) = C + S_p(I)$$

$S_p(I)$ → Variable space requirement

C → Fixed space requirement

a. Fixed space do not depend on the number and size of program inputs and outputs are considered.

   C = program space + data space + stack space

b. Variable space is the space required for the variables whose size depends on the particular instance of the problem considered. It includes additional space required when a function uses recursion

$S_p(I)$ = program space + data space + stack space
+ space used at recursion

Normally, we neglect (ignore) C and concerned with only variable space requirements.

$S(P) = S_p(I)$

Time complexity

The time efficiency of an algorithm is the amount of computer time required to run the program till the completion. Since the efficiency of an algorithm is measured with respect to time, this is often associated with an algorithm.

The time efficiency is calculated using step count i.e., how many times a program step is executed. The number of program steps or step count can be obtained using two methods.

a. Counting method : Use a global variable count with initial value of 0 and insert a statement that increment count by 1 for each executable statement.

```
float sum (float a[] , int n)
{
    float sum = 0.0 ;          count ++
    int i ;
    for(i = 0 ; i<n ; i++)     count ++
        sum = sum+a[i];        count ++
    return sum ;               count ++
}
```

Total count = 2n+3

b. Tabular method : The following procedure is used to obtain step count.

Step 1 : Determine the step count for each statement

Step 2 : Find out the number of times each statement is exe-cuted. This is called frequency.

Step 3 : Multiply step 1 and step 2 to get total steps for each statement.

Step 4 : Add the total's obtained in step 3 to get the step count for the entire function.

```
float sum (float a[], int n)
{
    float sum = 0.0;
    int i;
    for(i=0; i<n; i++)
        sum = sum+a[i];
    return sum;
}
```

s/e	frequency	total
0	0	0
0	0	0
1	1	1
0	0	0
1	n+1	n+1
1	n	n
1	1	1
0	0	0

Total steps = 2n+3

Asymptotic Notations

The notations we used to describe the asymptotic running time of an algorithm are defined in term of functions whose domains are the set of natural numbers.

$N = \{0, 1, 2, \ldots\}$, such notations are convenient for describing the worst-case running time function $T(n)$, which is usually defined on integer input sizes.

Different kinds of notations are :

a. O - BigOh

b. Θ - BigTheta

c. Ω - BigOmega

d. o - LittleOh

Using asymptotic notations we can measure the algorithm performance

Best case : The efficiency of an algorithm for the input of size n for which an algorithm takes atleast less time during execution among all possible input of size n is called Best Case efficiency.

Worst case : The efficiency of an algorithm for the input of size n for which an algorithm takes maximum time during execution among all possible input of size n is called Worst Case efficiency.

Average case : The efficiency of an algorithm for the input of size n for which an algorithm takes average time during execution among all possible input of size n is called Average Case efficiency.

Example : Linear Search Algorithm

Best - Element in 1^{st} position

Worst - Last position / Absent

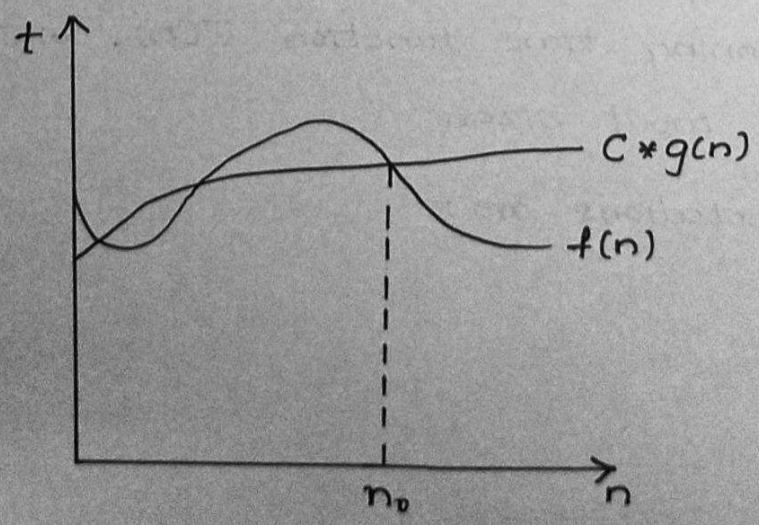
Average - Between 1^{st} and last position

Big-Oh (O)

It is a formal method of expressing our algorithm in upper bound, consumes maximum amount of time.

Let, $f(n)$ be the efficiency of an algorithm, $f(n)$ is said to be big-oh of another function $g(n)$, such that their exist a positive constant C and positive integer n_0 satisfying the following constraint.

$$f(n) \leq C * g(n) \quad \text{for } n \geq n_0$$



Steps

* Take the lower order term of $f(n)$, replace the constant with the next higher order variable, and call it as $C * g(n)$
* Once the constraint $f(n) \leq C * g(n)$ for $n \geq n_0$ is obtained we say $f(n) \in O(g(n))$

Ex: $f(n) = 100n + 5$ express $f(n)$ using Big-Oh

$= 100n + 5 \Rightarrow 100n + n = 101n$

$C * g(n) = 101n$ where $C = 101$, $g(n) = n$

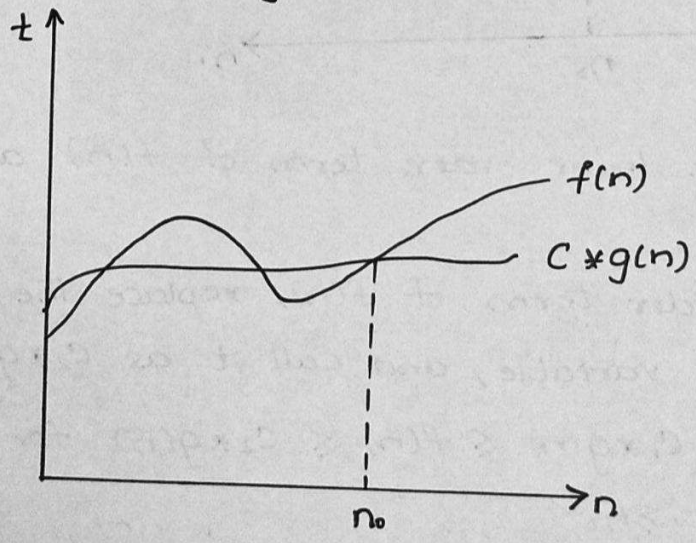
$f(n) \leq C * g(n)$ for $n \geq n_0$

for $n \geq 5$ $f(n) \in O(g(n))$

Big-Omega (Ω)

Let, $f(n)$ be the efficiency of an algorithm, $f(n)$ is said to be big-omega of another function $g(n)$, such that their exist a positive con-stant C and positive integer n_0 satisfying the following constraint.

$f(n) \geq C * g(n)$ for $n \geq n_0$



Steps

* Remove the lower order term of $f(n)$ and call it $C * g(n)$
* Once the constraint $f(n) \geq C * g(n)$ for $n \geq n_0$ is obtained we say $f(n) \in \Omega(g(n))$

Ex: $f(n) = 100n^3 + 5$ express $f(n)$ using Big-Omega

$= 100n^3 + 5 \Rightarrow 100n^3$

$C * g(n) = 100n^3$ where $C = 100$, $g(n) = n^3$

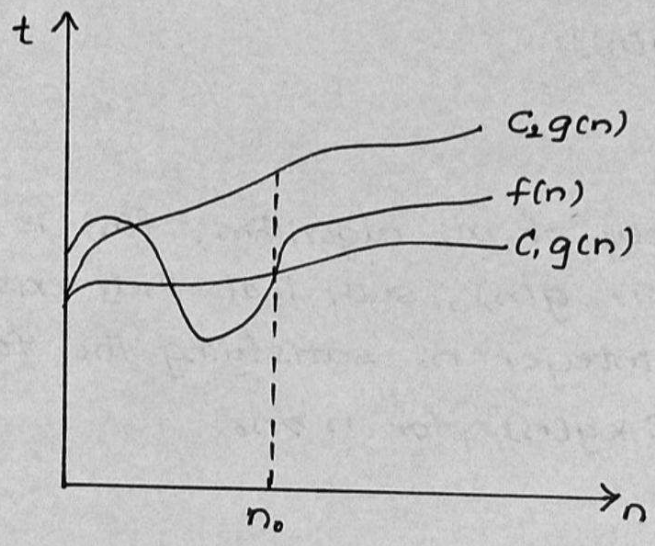
$f(n) \geq C * g(n)$ for $n \geq n_0$

Big-Theta (Θ)

Let, $f(n)$ be the efficiency of an algorithm, $f(n)$ belongs to the set $\Theta(g(n))$ if their exist positive constant C_1 and C_2 such that it can be "sandwitched" between $C_1 g(n)$ and $C_2 g(n)$ for large n.

$f(n) \in \Theta(g(n))$ or $f(n) = \Theta(g(n))$ and satisfying

$0 \leq C_1 * g(n) \leq f(n) \leq C_2 * g(n)$ for all $n \geq n_0$ such that their exist positive constant C_1 and C_2.



Steps

* Remove or ignore the lower order term of $f(n)$ and call it as $C_1 * g(n)$.
* Consider the lower order term of $f(n)$, replace the constant with the next higher order variable, and call it as $C_2 * g(n)$.
* Once the constraint $C_1 * g(n) \leq f(n) \leq C_2 * g(n)$ for $n \geq n_0$ we say $f(n) \in \Theta(g(n))$ for $n \geq n_0$

Ex: $f(n) = 100n + 5$

$C_1 * g(n) = 100n$ where $C_1 = 100$, $g(n) = n$

$C_2 * g(n) = 101n$ where $C_2 = 101$, $g(n) = n$

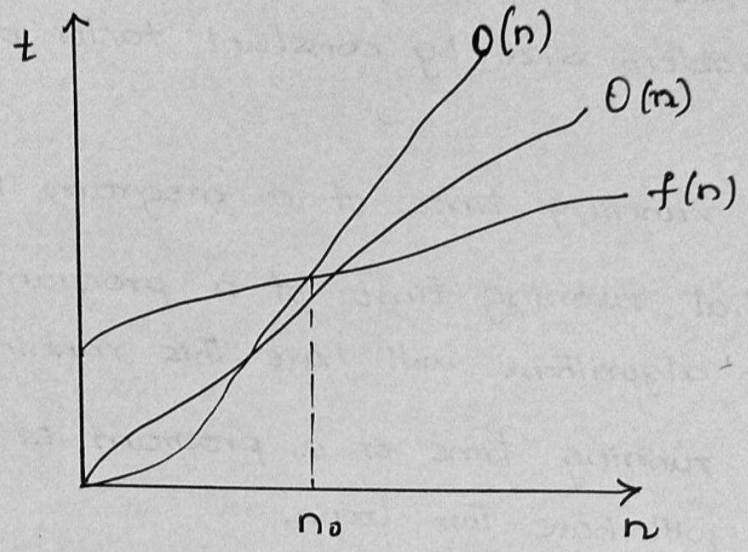
$f(n) \in \Theta(g(n))$ for $n \geq n_0$

$n \geq 5$.

Little - oh Notation

Let, $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers.

We say that $f(n)$ is $o(g(n))$ or $f(n) \in o(g(n))$ if for any real constant $n_0 \geq 1$ such that $0 \leq f(n) < c * (g(n))$.



In mathematical relation, $f(n) = o(g(n))$ means

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

It means little o() means loose upper-bound of $f(n)$.

Practical Complexities

The practical complexity of an algorithm is normally expressed as a function of n. The order of n may be 0, 1 or 2 and so on. The behavior of algorithm changes as the order of n changes.

1: Indicates that running time of a program is constant.

log N: Indicates that running time of a program is logarithmic. This running time occurs in programs that solves larger problems by reducing the problem size by constant factor at each iteration of loop.

N: Indicates that running time of a program is linear.

NlogN: Indicates that running time of a program is NlogN. The divide and conquer algorithms will have this running time.

N^2: Indicates that running time of a program is quadratic. The algorithms normally will have two loops.

N^3: Indicates that running time of a program is cubic. The algorithms normally will have three loops.

2^N: Indicates that running time of a algorithm is exponential. The algorithm that generate subsets of a given set will have this running time.

N!: Indicates that running time of an algorithm is factorial.

Units of measuring the time

Count the number of times an algorithms basic operation is executed. It is the most time consuming operation in the algorithm

Ex: The basic operation is the most time consuming operation in the algorithm. innermost loop.

* Let C be the execution time of a basic operation is executed.
* Let C(n) be the total number of times the basic operation is executed.

Then the running time T(n) is given by,

$$T(n) \approx C * C(n)$$

This T(n) varies linearly with the increase or decrease in the value of n.

Order of Growth

The change in the behavior as the value of n increases is called order of growth. Here we consider only the leading term of a formula and ignore the constant co-efficient.

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	n!
10	3.3	10	$3.3 \cdot 10$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{57}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		

Mathematical Analysis of Non-Recursive Algorithms

* Decide on a parameter indicating an input's size.
* Identify the algorithms basic operation.
* Check whether the number of times basic operation is executed depends only on the inputs size n or by any additional property is added to this.
* Set up summation for C(n) reflecting the number of times the algorithms basic operation is executed.
* Simplify summation using standard formulas.

Mathematical Analysis of Recursive Algorithms

* Decide on a parameter indicating an inputs size.
* Identify the algorithms basic operation.
* Check whether the number of times basic operation is executed depends only on the input size n.
* Set up recurrence relation, with an appropriate initial condition, for the number of times the algorithms basic operation is exeauted.
* Solve the recurrence relation.