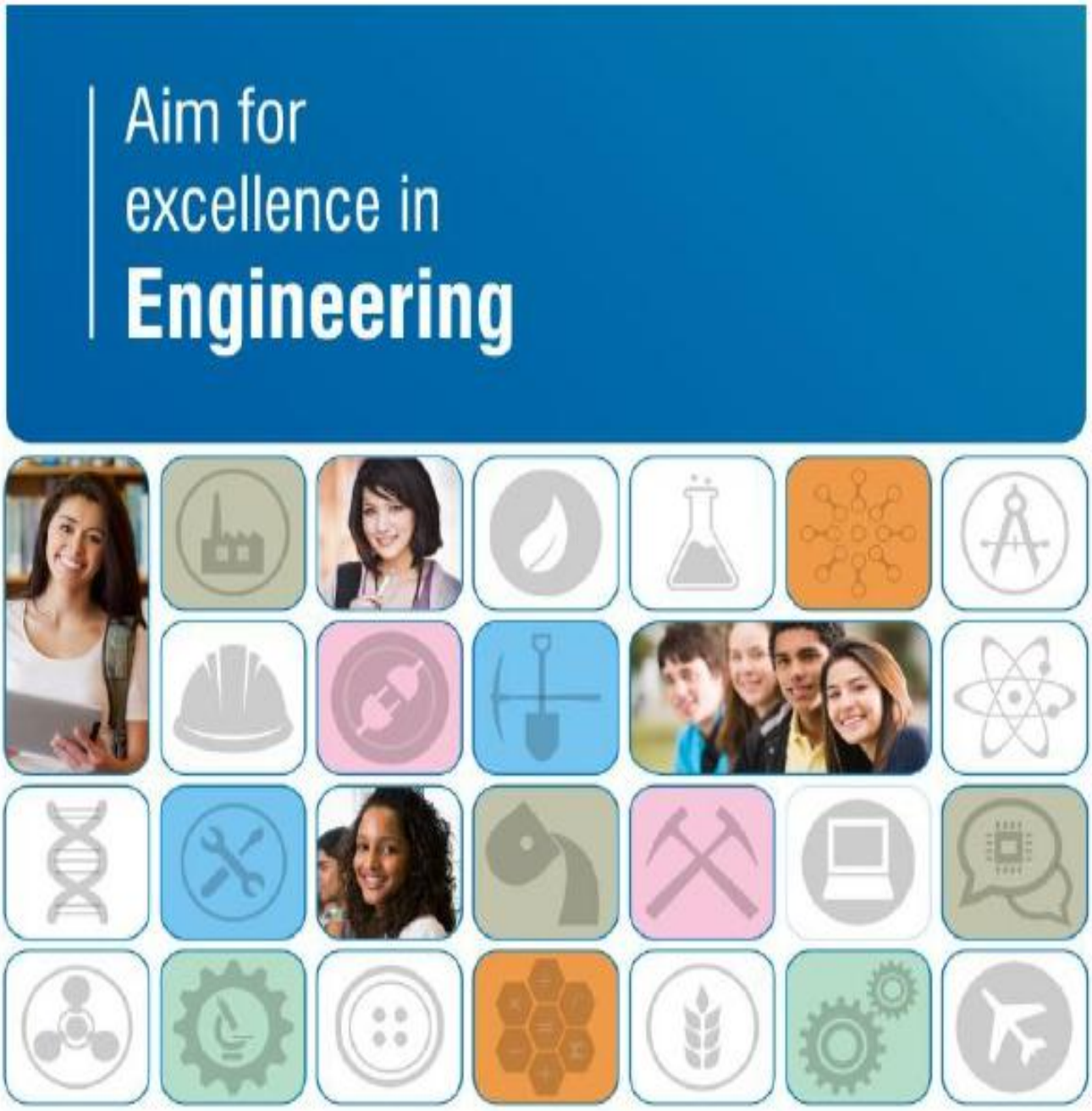


COMPUTER SCIENCE

Algorithms

CAREER AVENUES

Algorithms

INTRODUCTION TO ALGORITHMS

An algorithm is a set of well-defined steps required to accomplish some task. If you've ever baked a cake, or followed a recipe of any kind, then you've used an algorithm. Algorithms also usually involve taking a system from one state to another, possibly transitioning through a series of intermediate states along the way.

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

An algorithm is a tool for solving a well-specified computational problem. For example, one might need to sort a sequence of numbers into non-decreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the sorting problem:

Input: A sequence of n numbers a1, a2, ..., an.

Output: A permutation (reordering) (a'1.a'2……a'n) of the input sequence such that a'1<=a'2<=….a'n.

PROPERTIES OF ALGORITHM

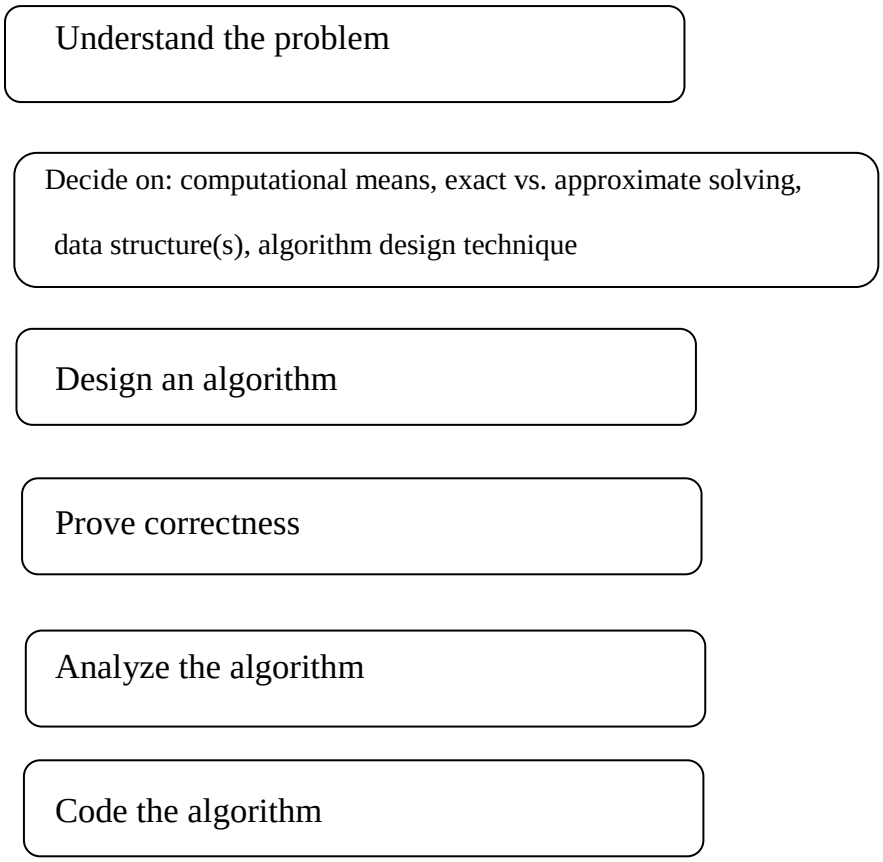
An algorithm has the following five basic properties

- A number of quantities are provided to an algorithm initially before the algorithm begins. These are the inputs that are processed by the algorithm.
- The processing rules specified in the algorithm must be precise, unambiguous and lead to a specific action. An instruction which can be carried out is called an effective instruction.
- Each instruction must be sufficiently basic such that it can be carried out in a finite time by a person with paper and pencil.
- The total time to carry out all the steps in the algorithm must be finite.
- An algorithm must have one or more output.

FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

- Algorithms are considered to be procedural solutions to problems.
- The solutions are not answers to the problems but specific instructions for getting the answers.

ALGORITHM DESIGN & ANALYSIS PROCESS



ALGORITHM DESIGN TECHNIQUE

- It is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Reasons to study different techniques
    - They provide guidance for designing algorithms for new problems.
    - Problems that do not have satisfactory solutions.
    - Algorithm design technique makes it possible to classify algorithms according to an underlying idea.
    - They serve as a natural way to categorize and study the algorithms

METHODS OF SPECIFYING AN ALGORITHM

- Using Natural Language
    - This method leads to ambiguity.
    - Clear description of algorithm is difficult.

- Using Pseudo codes
    - It is a mixture of natural language and programming language like constructs.
    - It is more precise than a natural language.
    - Earlier days we used another method.
    - Using Flow Charts
    - It is a method of expressing an algorithm by a collection of connected geometric shapes containing the description of algorithm.

ANALYZING AN ALGORITHM

- Two kinds of algorithm efficiency
    - Time efficiency
        - How fast the algorithm runs.
    - Space efficiency
        - How much extra space the algorithm needs.
    - Other desirable characteristics
        - Simplicity
            - Simpler algorithms are easier to understand.
            - It depends on the user.
        - Generality
            - Two issues
            - Generality of the problem the algorithm solves.
            - Range of inputs.

IMPORTANT PROBLEM TYPES

SORTING

It refers to rearranging the items of a given list in ascending order. For example,

- Sort numbers, characters, strings, records.

We need to choose a piece of information to be ordered.

- This piece of information is called a key.
- The important use of sorting is searching.
- There are many algorithms for sorting.
- Although some algorithms are indeed better than others but there is no algorithm that would be the best solution in all situations.

- The two properties of sorting algorithms are
    - Stable
    - In place
- A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
- An algorithm is said to be in place if it does not require extra memory, except possibly for a few memory units.

SEARCHING

- It deals with finding a given value called search key, in a given set.
    - There are several algorithms ranging from sequential search to binary search.
    - Some algorithms are based on representing the underlying set in a different form more conductive to searching.
    - They are used in large databases.
    - Some algorithms work faster than others but require more memory.
    - Some are very fast only in sorted arrays.

STRING PROCESSING

A string is a sequence of characters. We are interested in three kinds of strings

- Text strings
    - Comprises of letters, numbers and special characters
- Bit strings
    - Comprises of zeroes and ones.
- Gene sequences
    - Modeled by strings of characters from the four character alphabet A, C, G, T
- String processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.
- String matching is one kind of such problem.

GRAPH PROBLEM

- A graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.

- They can be used for modeling wide variety of real life applications.
- Basic graph algorithm includes
- Graph traversal algorithms
- Shortest path algorithms
- Topological sorting for graphs with directed edges.

COMBINATORIAL PROBLEMS

These problems ask to find a combinatorial object such as a permutation, a combination, or a subset – that satisfies certain constraints and has some desired property. These are the most difficult problems.

Reasons

- The number of combinatorial objects grows extremely fast with a problem's size reaching unimaginable magnitude even for moderate sized instances.
- There are no algorithms for solving such problems exactly in an acceptable amount of time.

GEOMETRIC PROBLEMS

They deal with geometric objects such as points, lines, and polygons.

These algorithms are used in developing applications for computer graphics, robotics.

The method is used in radiology, archaeology, biology, geophysics, oceanography, materials science, astrophysics and other sciences.

NUMERICAL PROBLEMS

These are the problems that involve mathematical objects of continuous nature:

- Solving equations, system of equations, computing definite integrals, evaluating functions.
- The majority of such problems can be solved only approximately.
- Such problems require manipulating real numbers, which can be represented in computer only approximately.
- Large number of arithmetic operation leads to round off error which can drastically distort the output.

ANALYSIS OF ALGORITHMS

There may be many different ways to solve a given problem, i.e. there may be many possible algorithms. Given a choice, which of the algorithms should be chosen? What are the possible reasons for choosing one algorithm over another?

We would like to choose the "better" algorithm. But what does it mean for an algorithm to be better? To answer this question we need to 'analyse' the algorithms at hand. Analysis of an algorithm is meant to predict the amount of resources it requires. In computing terms, one of the important resources is 'time'. If an algorithm takes lesser time to complete a task, it would generally be considered the 'better' algorithm. These notions will be formalized mathematically.

As an example, consider the first problem that was mentioned: sorting. Given n numbers, a1, a2, …. an, we need to sort them in ascending order. A simple algorithm and its analysis is given below. The notation used will be the array notation a[1], a[2] and so on.

```
For i = 1 to n do
        pos ← i                              n times
        For j = i+1 to n do                  loop is run (n – i) times
                If (a[j] < a[pos])           (n – i) times in every loop
                        pos = j              (n – i) times in every loop (possibly)
        tmp ← a[i]                           n times
        a[i] ← a[pos]                        n times
        a[pos] ← tmp                         n times
```

In this algorithm (selection sort) we loop through the list and find the smallest element and move it to the start of the array. Then we look for the smallest element from the second position onwards, and so on. Assuming that every step takes the same amount of time, some constant 'c', the total running time could be expressed as:

$$T(n) = c\,[n + 2 \sum_{j=2}^{n} (n - j + 1) + 3n] = c[4n + 2\,n(n-1)/2] = 3cn + cn^2.$$

Here we have considered the worst case, when the condition in the inner loop is satisfied every time. So in every case the number of steps required will be at max $3n + n^2$. Also note, that we have not counted the operation of incrementing i and j. Adding these will result in a change in the coefficients of n and n^2.

What happens as we run the algorithm on large arrays, i.e. the value of n is large? Of the two terms in the polynomial, the term with a lower degree becomes more or less insignificant as n increases. That is to say, the square term, n^2 (in this case) becomes more important, and for getting a practical idea of the running time within reasonable error limits, we may ignore the 3cn term altogether. This idea is formalized in the next section on asymptotic notations.

ASYMPTOTIC NOTATION

As noted in the previous section, all terms other than the leading term may be ignored when judging the efficiency of an algorithm. We can also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. Thus, from $cn^2 + 3cn$, we can get an abstraction of n^2. We say that the running time of the selection sort algorithm is $O(n^2)$. When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

THETA NOTATION (IN BOUND) Θ

- Θ (g(n)) is the set of functions f(n) such that there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$ [1]

- "$f(n) \in \Theta(g(n))$"
- "$f(n) = \Theta(g(n))$"
- g(n) is an asymptotically tight bound of f(n)
- Θ notation bounds a function within constant factors

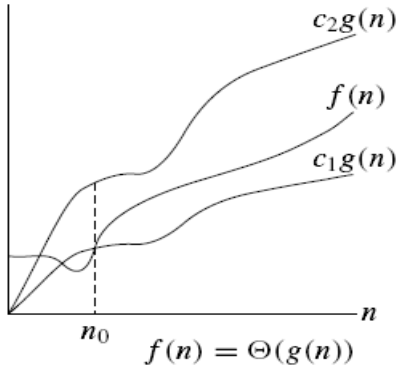
Example:

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for all $n \geq n_0$. Dividing by n^2 yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 .$$



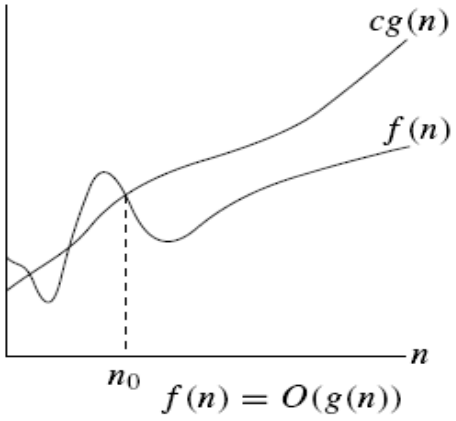
(a)

BIG 'O' NOTATION (UPPER BOUND) O

- f(n) = O(g(n)) iff there exist positive constants c and n_0 such that $0 \le f(n) \le c*g(n)$ for all $n \ge n_0$

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

- $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$
- $\Theta(g(n)) \subseteq O(g(n))$.
- O gives an upper bound for a function within a constant factor.
- f(n) = O(g(n)) some constant multiple of g(n) is an asymptotic upper bound of f(n),no claim about how tight an upper bound is.



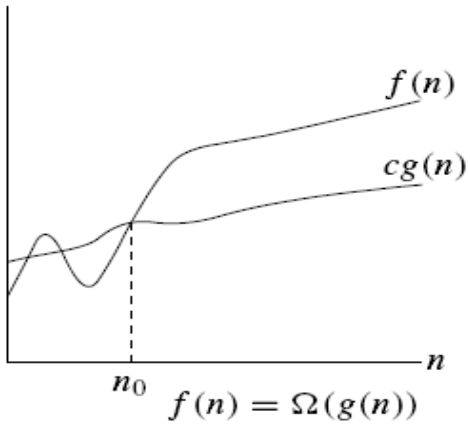
NOTE: Different choice of c gives different values of n_0. Example: $2n = O(n^2)$ as $2n <= 1.n^2$ for all n>=2(so here c=1 & n_0 =2)

OMEGA NOTATION (LOWER BOUND) Ω

- Ω (g(n)) is the set of functions f(n) such that there exist positive constants c and n_0 such that $0 \le c*g(n) \le f(n)$ for all $n \ge n_0$

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

- Ω gives a lower bound for a function within a constant factor.



Ω gives a lower bound for a function within a constant factor.

Example: $3n^2 = \Omega(n)$ as $3.n^2 >= 3.n$ for all n >= 1 (so here c = 3 and n_0 =1)

Theorem:

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

Small o-Notation :

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\}.$$

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

Small Omega w-Notation :

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\} .$$

$$f(n) \in \omega(g(n)) \text{ if and only if } g(n) \in o(f(n)) .$$

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

Properties:

Transitivity:

$$
\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) \\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) \\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) \\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) \\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n))
\end{aligned}
$$

Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n)) , \\
f(n) &= O(f(n)) , \\
f(n) &= \Omega(f(n)) .
\end{aligned}
$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$
\begin{aligned}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) , \\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .
\end{aligned}
$$

Analogy of asymptotic notations to comparison of two real numbers, a, b.

$$
\begin{aligned}
f(n) = O(g(n)) &\approx a \leq b , \\
f(n) = \Omega(g(n)) &\approx a \geq b , \\
f(n) = \Theta(g(n)) &\approx a = b , \\
f(n) = o(g(n)) &\approx a < b , \\
f(n) = \omega(g(n)) &\approx a > b .
\end{aligned}
$$

Logarithms Equality:

For all real numbers a > 0, b > 0, c > 0 and n

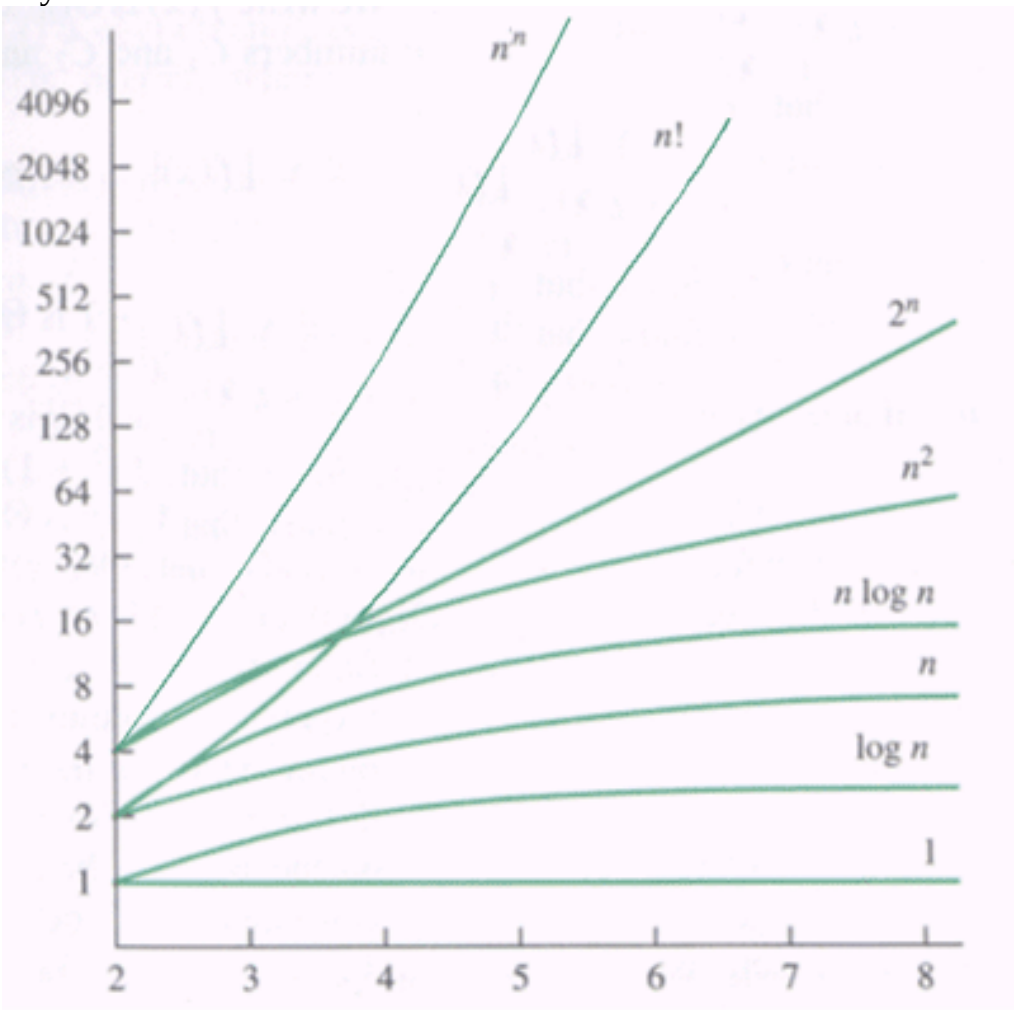
$$
\begin{aligned}
a &= b^{\log_b a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n \log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b}, \\
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b c} &= c^{\log_b a},
\end{aligned}
$$

TIME COMPLEXITY

The total number of steps involved in a solution to solve a problem is the function of the size of the problem, which is the measure of that problem's time complexity. Some general order that we can consider

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(c^n) < O(n!)$, where c (>=2) is some constant.

Complexity classes:



SPACE COMPLEXITY

Space complexity is measured by using polynomial amounts of memory, with an infinite amount of time.

The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or non-determinism. Amount of computer memory required during the program execution, as a function of the input size.

A small amount of space, deterministic machines can simulate nondeterministic machines, where as in time complexity, time increase exponentially in this case. A nondeterministic TM using O(n) space can be changed to a deterministic TM using only $O(n^2)$ space. Generally, the efficiency of an algorithm is judged based on the time complexity, rather than space complexity for two reasons:

- Firstly time is a more valuable resource as far as computing is concerned. Space (or storage) is actually cheap.
- Secondly, most of the algorithms that are generally used do not have large space requirements. Most of them actually require space of the order O(n). So there is not much difference between the various algorithms, as far as space complexity is concerned.

Due to these reasons, when the term "efficient algorithm" is used, we generally mean a lower time complexity.

WORST CASE AND AVERAGE CASE

The running time of an algorithm may vary depending on the type of input provided to it. For instance, there are sorting algorithms whose running time depends on the type of array provided. Their time complexity may vary depending on the initial arrangement of the numbers in the array. For instance the simple quick sort algorithm is not very efficient for sorting if the array happens to be initially sorted in the reverse order. But in general practical cases, quick sort is found to be efficient.

Therefore, for many algorithms, two different types of analysis need to be made. The first is the worst case, which is a measure of the worst performance possible over the set of all possible inputs. The second is the average case, which is based on a probabilistic analysis of the various types of input possible. If the algorithm has a high worst case complexity, but a good average case complexity, it may still be a good option, because the particular worst case inputs that deteriorate the algorithm's performance are not expected to be encountered very often.

When the worst case and average complexities are quite different, they will be mentioned separately as and when new algorithms are encountered. One may also consider the 'best-case', i.e. time complexity for inputs for which the algorithm runs the fastest. However, this is not of much practical significance since the best case is not expected to be encountered on a regular basis.

Questions

1. Is $2^{n+1} = O(2^n)$

2. Is $2^{2n} = O(2^n)$

3. Show that for constants a and b, b> 0, $(n + a)^b = \Theta(n^b)$

Solutions

1. Here $f(n) = 2^{n+1} = 2.\ 2^n$, $g(n) = 2^n$. Taking $c = 2$ and $n_0 = 1$, we have $f(n) = 2^{n+1} = 2.\ 2^n \leq 2.\ 2^n = c\ g(n)$ for $n \geq n_0$. Therefore $2^{n+1} = O(2^n)$.

2. Suppose $2^{2n} = O(2^n)$. Then there exist c and n_0 such that for all $n \geq n_0$ $2^{2n} \leq c\ 2^n$.

   Taking lg on both sides, we get $2n \leq \lg c + n$, i.e. $n \leq \lg c$. Since the LHS will go on increasing continuously, there is no value of c that will satisfy the relation for all $n \geq n_0$.

   Therefore $2^{2n} \neq O(2^n)$

3. We need to find c_1, c_2, and n_0 such that for all $n \geq n_0$,

   $$c_1\ n^b \leq (n + a)^b \leq c_2\ n^b.$$

   If $a \geq 0$, choose $c_1 = 1$ to satisfy the first inequality. Now for all $n \geq a$, $(n+a)^b \leq (2n)^b = 2^b n^b$. Therefore choosing $n_0 = a$, and $c_2 = 2^b$ satisfies the relation.
   If $a \leq 0$, choose $c_2 = 1$ to satisfy the second inequality. Now for all $n \geq -2a$, $(n/2) \geq -a$. Adding $(n/2)$ on both sides we get, $n \geq (n/2) - a$. Rearrange this to get $n+a \geq (n/2)$. So $(n+a)^b \geq (n/2)^b = 2^{-b} n^b$. Therefore choosing $n_0 = -2a$, $c_1 = 2^{-b}$ satisfies the first relation as well.

   Therefore $(n + a)^b = \Theta(n^b)$ for all $b > 0$.