Asymptotic notation

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Asymptotic Growth of Functions

- Greek: *asumptotos* - not touching (*a* - not, *sumptotos* - intersecting)
- Math: curves that do not intersect, but come infinitely close

Why Asymptotic notation ?

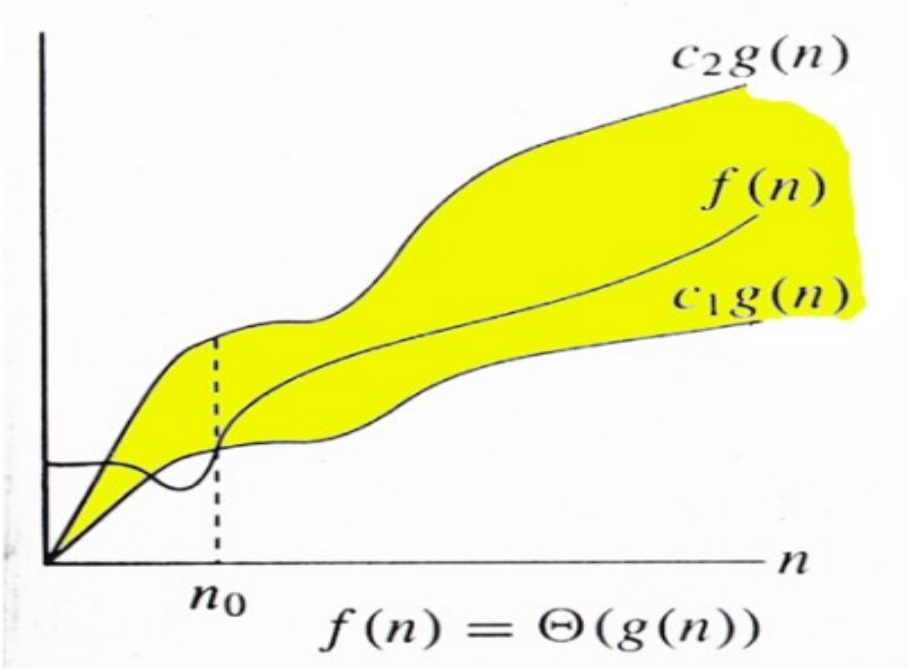
- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

Asymptotic notation

- Asymptotic notations are method used to estimate and represent the efficiency of an algorithm using simple formula.
- This can be useful for separating algorithms that leads to different amounts of work for large inputs.
- The notations we use to describe the asymptotic running time of an algorithm are defined in terms 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 only on integer input sizes.

Asymptotic notation

- Asymptotic notations give a limit (bound) to the growth of a function. This bound could be **upper** or **lower** bound for a function.
- A bound is asymptotically tight if it bounds the function with a constant difference.



Asymptotic notation

Why asymptotic notation ?

Order of growth

- Lower order item(s) are ignored, just keep the highest order item.
- The constant coefficient(s) are ignored.
- The rate of growth, or the order of growth, possesses the highest significance.
- Use $\Theta(n^2)$ to represent the worst case running time for insertion sort.
- Typical order of growth: $\Theta(1)$, $\Theta(\lg n)$, $\Theta(\sqrt{n})$,$\Theta(n)$, $\Theta(n\lg n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(2^n)$, $\Theta(n!)$
- Asymptotic notations: Θ, O, Ω, o, ω.

Asymptotic Complexity

- Running time of an algorithm as a function of input size n **for large n**.
- Expressed using only the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say $\Theta(n^2)$.
- Describes behavior of function in the limit.
- Written using ***Asymptotic Notation***.

Asymptotic Notation

- $\Theta, O, \Omega, o, \omega$
- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

4. Asymptotic Growth Rates

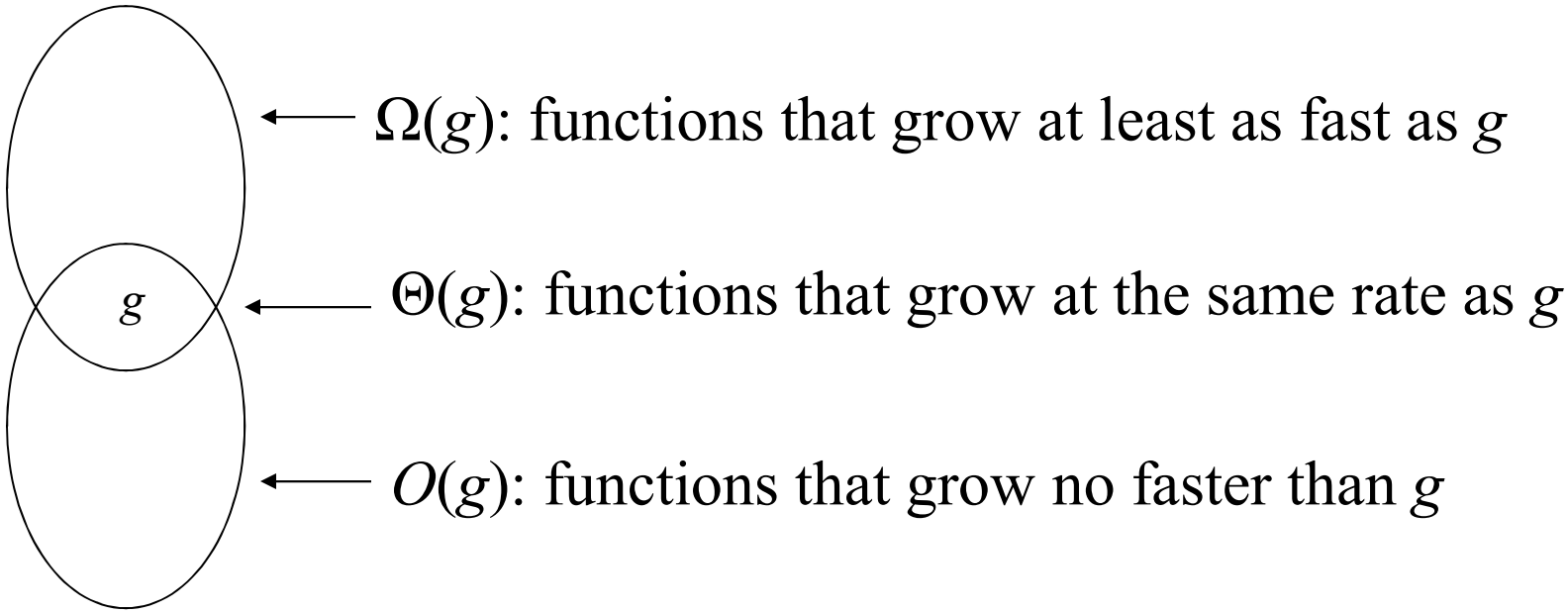
- "Work done": how good is this measure?
  - We do not count every step
- Suppose we have an algorithm with $2n$ basic operations and another with $4.5n$
  - $2cn$ and $4.5dn$ total operations
  - Which runs faster?
- If functions that describe the behavior differ by a constant factor, the algorithms are in the same *complexity class*

4. Asymptotic Growth ...contd

- Suppose we have an algorithm with $n^3/2$ basic operations and another with $5n^2$
  - Which runs faster?
  - For small n, the first is better; for large n, the second is better
- The *rate of growth* of a cubic function is much greater than that of a quadratic function
- $\Rightarrow$ ignore constant factors & small inputs

4. Asymptotic Growth ...contd

- Asymptotic notations
  - Big oh (O), big omega (Ω), big theta (Θ)



4. Asymptotic Growth ...contd

- For a given function $g(n)$, $\Theta(g(n))$, $\Omega(g(n))$, and $O(g(n))$ are sets of functions such that

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

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

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

4. Asymptotic Growth ...contd

- E.g., if f is in $O(g)$ we say, "f is oh of g"
- Suppose $f(n)=n^3/2$ and $g(n)=37n^2+120n+17$
  - Prove (a) g is in $O(f)$ and (b) f is not in $O(g)$

  (a) for $n \geq 78$, $g(n) \leq 1\, f(n) \Rightarrow g$ is in $O(f)$

  (b) Prove by contradiction that f is not in $O(g)$

- Lemma: f is in $O(g)$ if the limit of $f(n)/\, g(n)$ when $n \rightarrow \infty$ is $c < \infty$ (c can be 0)

4. Asymptotic Growth ...contd

- If f is in $\Theta(g)$ we say, "f is theta of g" or "f is order g" or "f is asymptotic order of g"
- Lemma: f is in $\Theta(g)$ if the limit of $f(n)/\ g(n)$ when $n \rightarrow \infty$ is c where $0 < c < \infty$
- Lemma: f is in $\Omega(g)$ if the limit of $f(n)/\ g(n)$ when $n \rightarrow \infty$ is > 0 (including ∞)

- Can use various theorems to compute limits, e.g., L'Hopital's Rule

4. Asymptotic Growth ...contd

- Example complexities
  - Searching an unordered array, finding the maximum element in an array
    - $\Theta(n)$ where n is the size of the array
  - Matrix multiplication algorithm above
    - $\Theta(n^3)$ where each matrix is nXn

BIG-OH COMPLEXITY

Big O notation

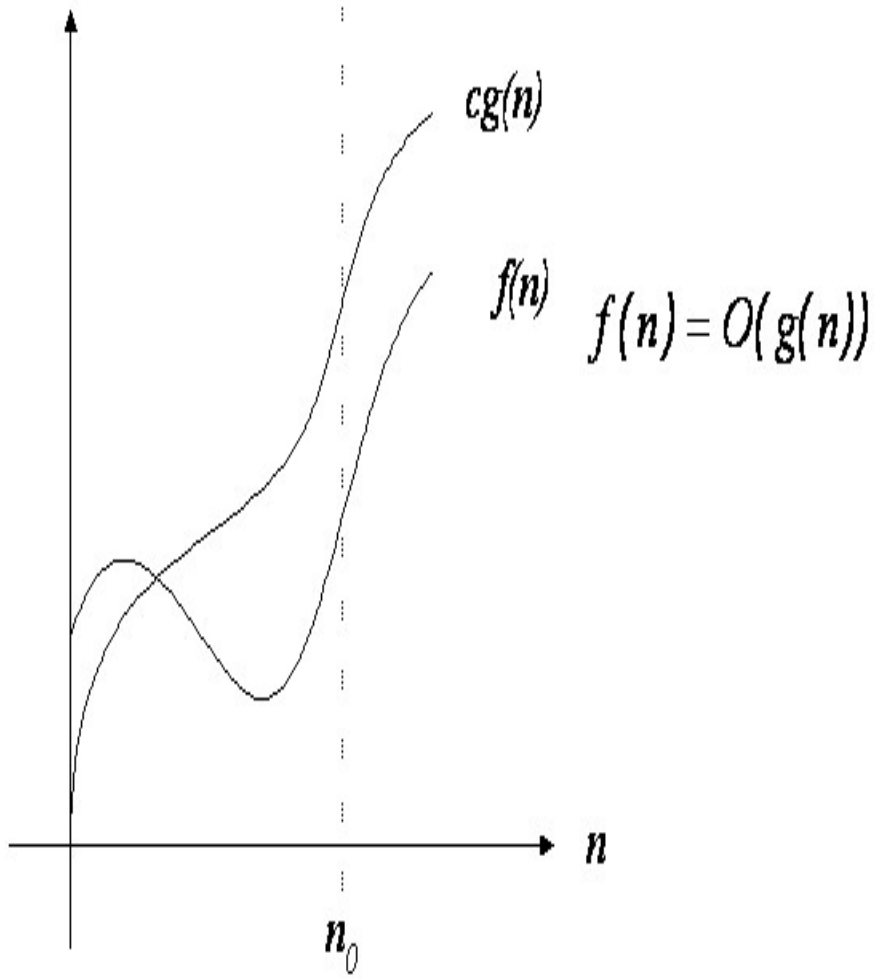
- Big O notation or Big Oh notation, and also Landau notation or asymptotic notation, is a mathematical notation used to describe the asymptotic behavior of functions.
- Its purpose is to characterize a function's behavior for very large (or very small) inputs in a simple but rigorous way that enables comparison to other functions.
- More precisely, the symbol *O* is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, usually simpler, function.

Big O notation

- There are also other symbols o, Ω, ω, and Θ for various other upper, lower, and tight bounds.
- Two main areas of application: in mathematics, it is usually used to characterize the residual term of a truncated infinite series, especially an asymptotic series, and in computer science, it is useful in the analysis of the complexity of algorithms.
- Informally, the O notation is commonly employed to describe an asymptotic tight bound, but tight bounds are more formally and precisely denoted by the Θ (capital theta) symbol as described below.

BIG Oh notation

This is a standard notation that has been developed to represent functions which *bound the computing time for algorithm* and it is used to define the worst case running time of an algorithm and concerned with very large values of ***n***.



BIG Oh notation

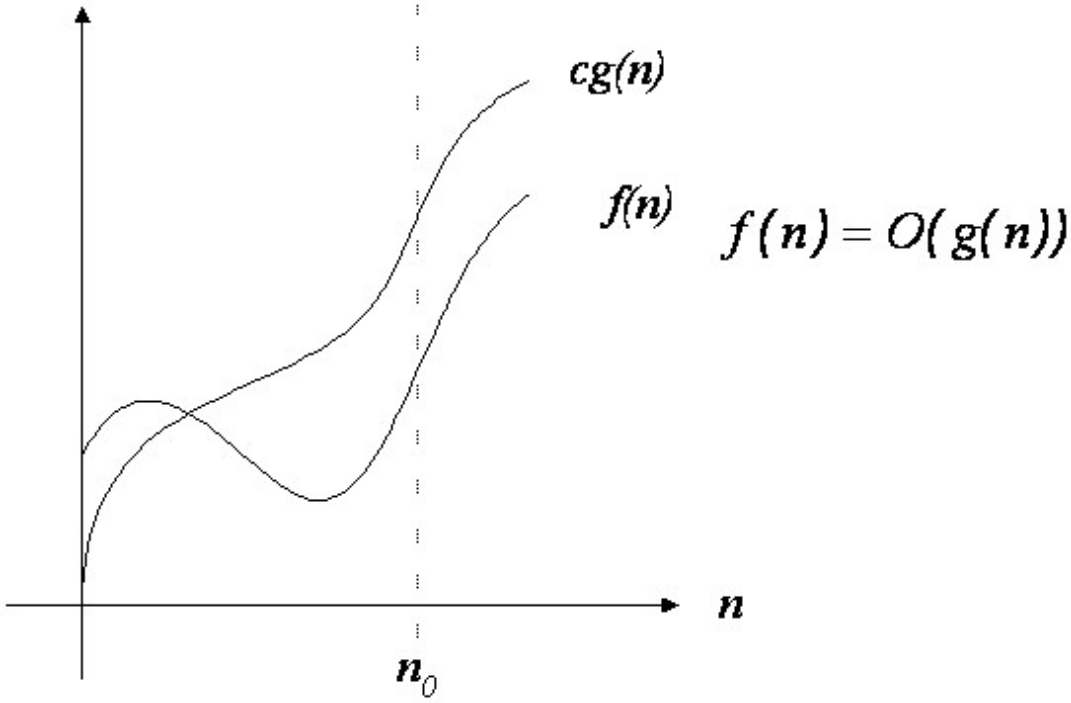
- Let f and g be functions from positive integers to positive integers. We say f is $O(g(n))$ (read: "f is order g") if g is an upper bound on f: there exists a fixed constant c and a fixed n_0 such that for all $n \geq n_0$,

$$f(n) \leq cg(n).$$

- Equivalently, f is $O(g(n))$ if the function $f(n)/g(n)$ is bounded above by some constant.

Big oh notation(O)

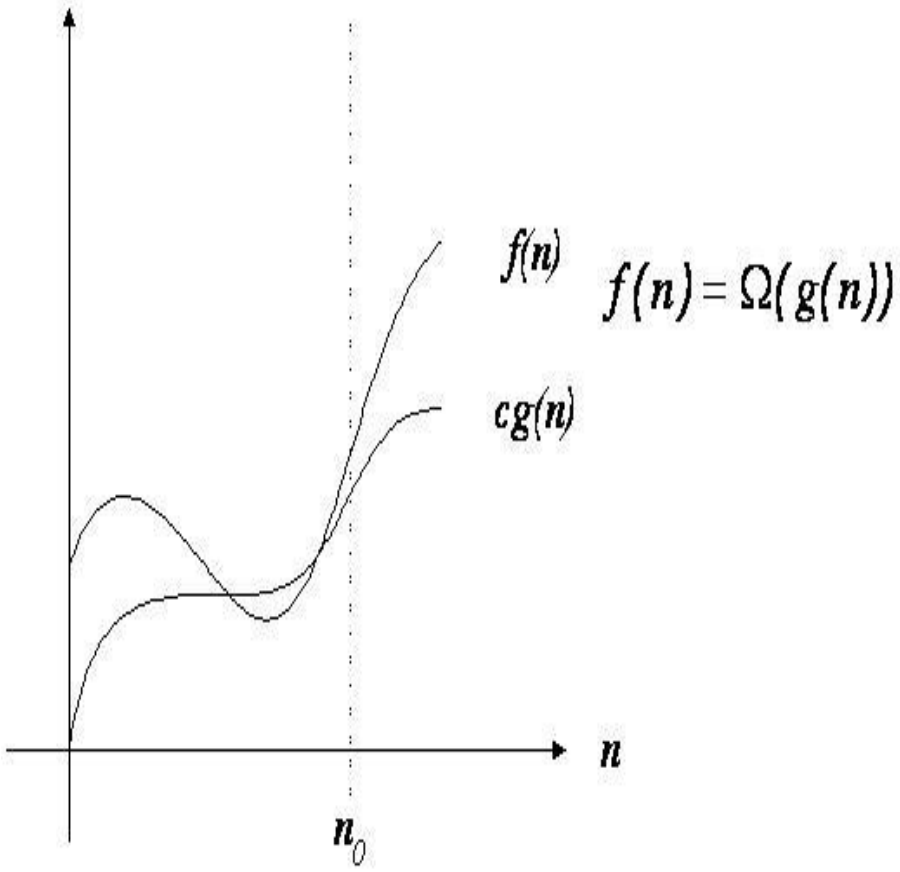
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$



Big oh notation(*O*)

- The meaning of an expression like $O(n^2)$ is really a set of functions: all the functions that are $O(n^2)$.
- When we say that $f(n)$ is $O(n^2)$, we mean that $f(n)$is a member of this set.
- It is also common to write this as $\boldsymbol{f(n) = O(g(n))}$ although it is not really an equality.

BIG OMEGA notation (Ω)



- This notation is used to describe the best case running time of algorithm and concerned with the very large values of n

BIG OMEGA notation (Ω)

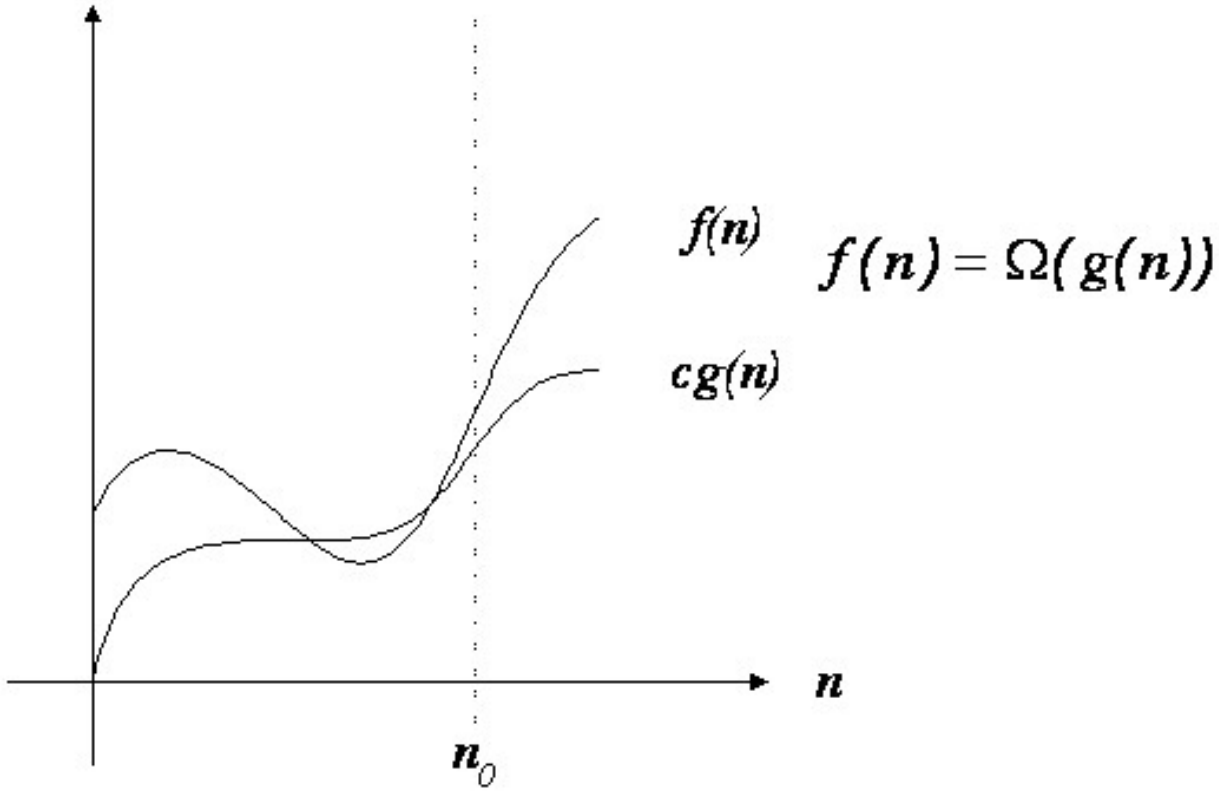
- We say that f is $\Omega(g(n))$ (read: "f is omega of g") if g is a *lower* bound on f for large n. Formally, f is $\Omega(g)$ if there is a fixed constant c and a fixed n_0 such that for all $n > n_0$,

$$cg(n) \leq f(n)$$

- For example, any polynomial whose highest exponent is n^k is $\Omega(n^k)$.
- If $f(n)$ is $\Omega(g(n))$ then $g(n)$ is $O(f(n))$. If $f(n)$ is $o(g(n))$ then $f(n)$ is *not* $\Omega(g(n))$.

Big omega notation (Ω)

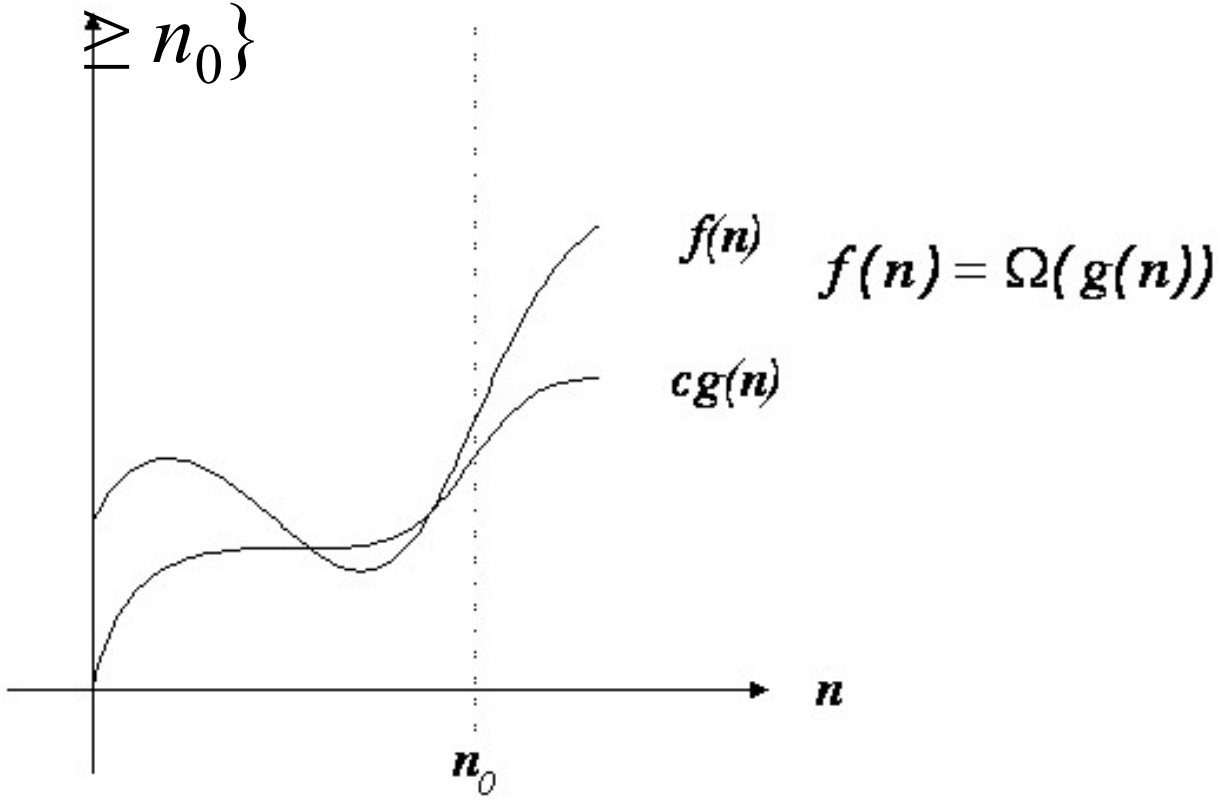
This notation is used to describe the best case running time of algorithm and concerned with the very large values of N.



$$f(n) = \Omega(g(n))$$

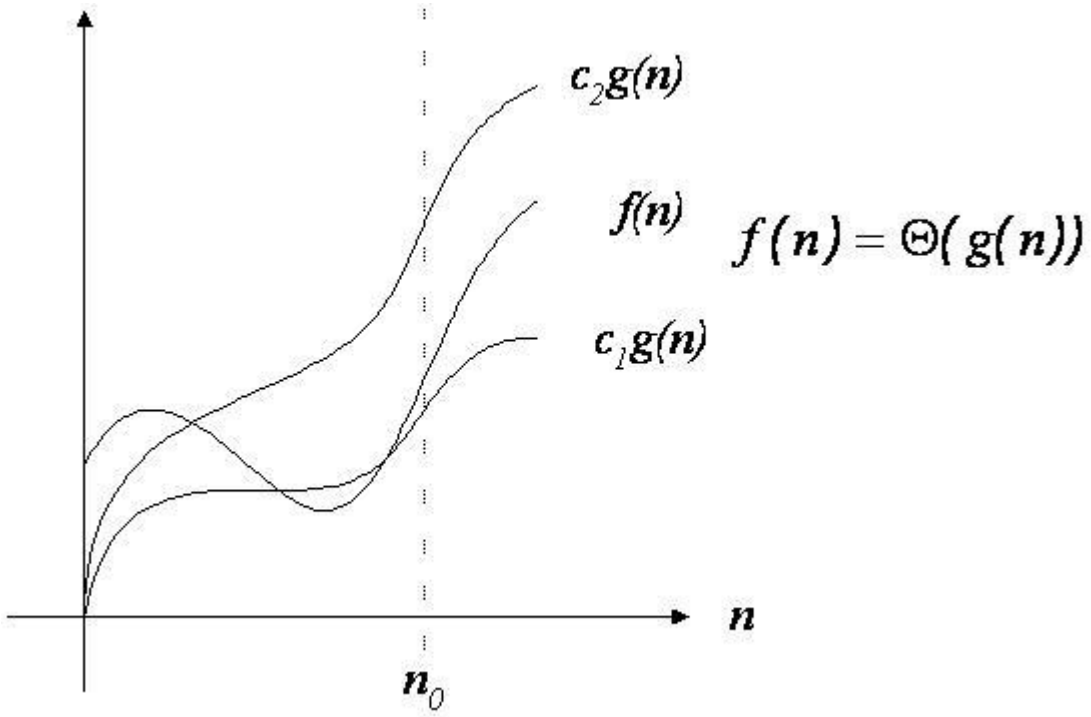
Defination: Big omega (Ω)

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



BIG THETA Notation

This Notation is used to describe the average case running time of algorithm and concerned with very large values of n.



Big theta notation(Θ)

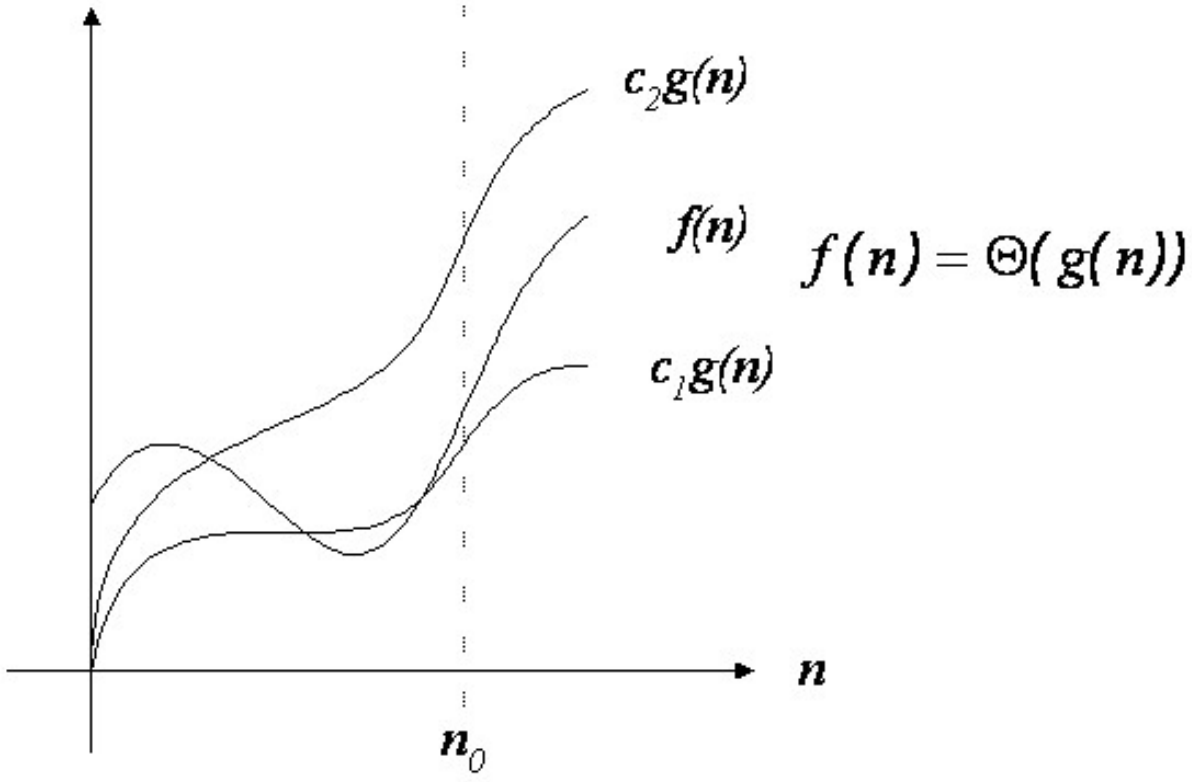
- We say that f is $\Theta(g(n))$ (read: "f is theta of g") if g is an accurate characterization of f for large n: it can be scaled so it is both an upper and a lower bound of f. That is, f is both $O(g(n))$ and $\Omega(g(n))$.
- Expanding out the definitions of Ω and O, f is $\Theta(g(n))$ if there are fixed constants c_1 and c_2 and a fixed n_0 such that for all $n > n_0$,

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

- For example, any polynomial whose highest exponent is n^k is $\Theta(n^k)$. If f is $\Theta(g)$, then it is $O(g)$ but not $o(g)$.
- Sometimes people use $O(g(n))$ a bit informally to mean the stronger property $\Theta(g(n))$; however, the two are different.

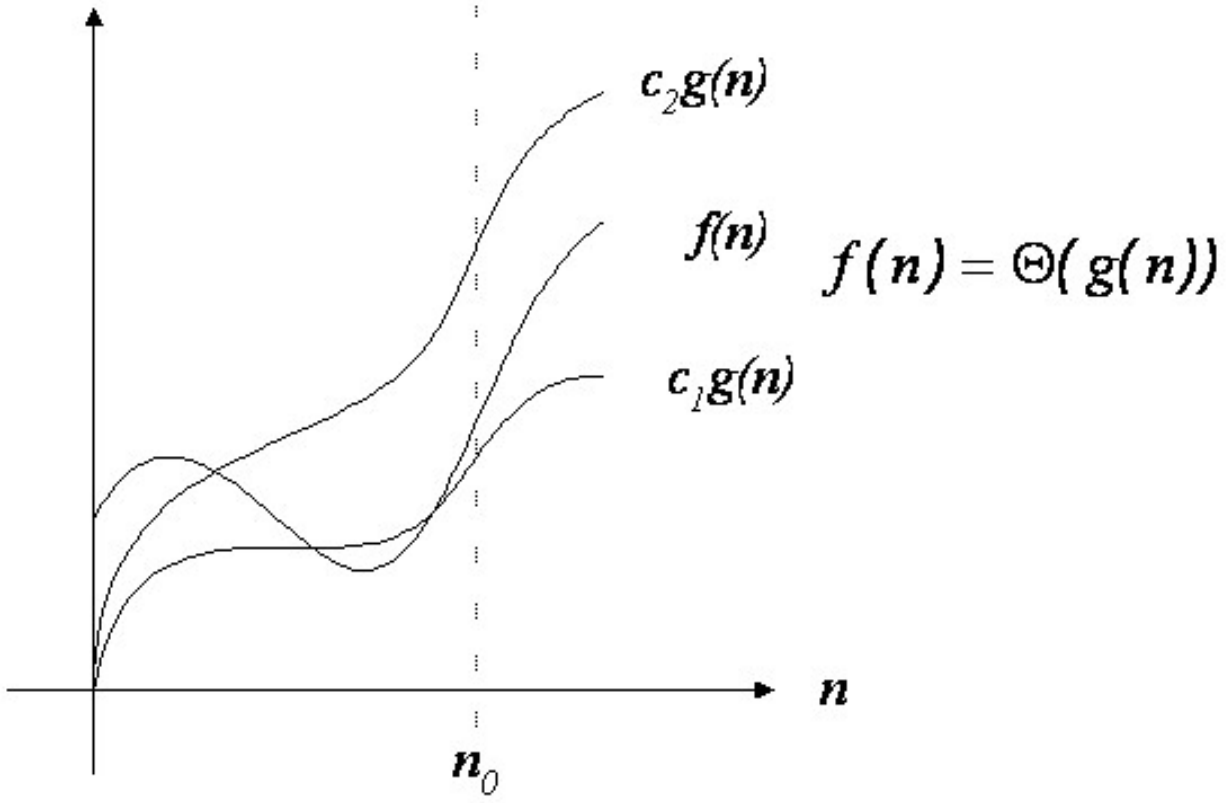
Big theta notation(Θ)

This Notation is used to describe the average case running time of algorithm and concerned with very large values of n.



Definition: Big theta (Θ)

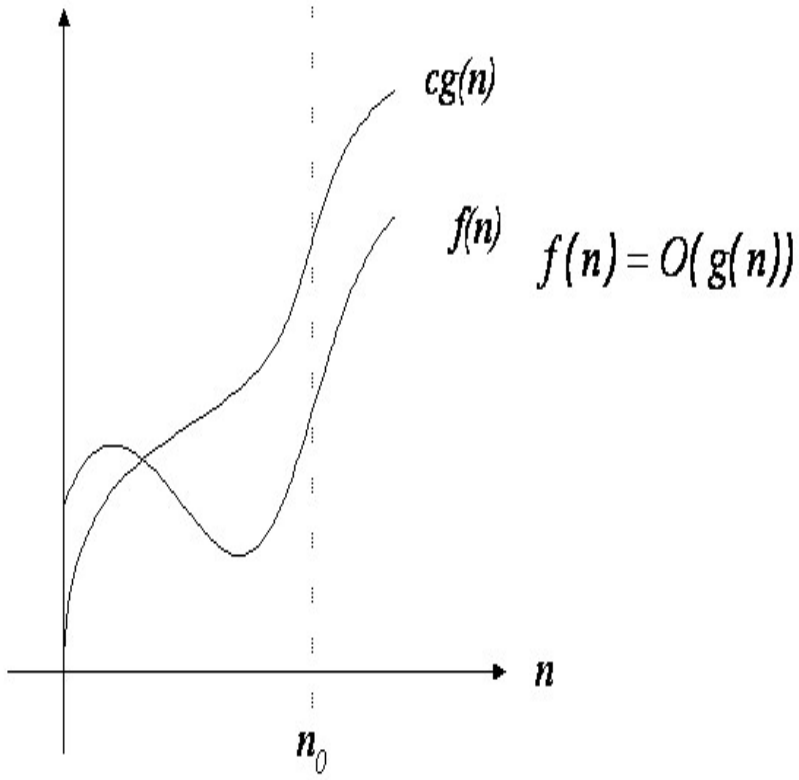
$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$



Calculating time complexity

- A simple program fragment

```
1. int
2. sum(int n)
3. {
4. int i, partialsum;
5. partialsum = 0;
6. for( i=1; i<= n; i++)
7. partialsum + = i*i*i;
8. return partialsum
9. }
```



6n+4 = O(n)

General rules

- *FOR loops*: the running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.
- *Nested FOR loop*: The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the size of all the for loops.
- Consecutive statements: Just add (which means that the maximum is the one that counts
- IF/ELSE: The running time of an If /else statement is never more than the running time of the test plus the larger of the runnning times of S1 and S2.
  - **If (condition)**
  - **S1**
  - **Else**
  - **S2**

Recall the Hierarchy of Growth Rates

Easy

$$c < \log^k N < N < N \log N < N^2 < N^3 <$$

$$2^N < 3^N < N! < N^N$$

Hard

We can make a distinction between problems that have polynomial time algorithms and those that have algorithms that are worse than polynomial time.

Limits

- $\lim_{n\to\infty}[f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty}[f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty}[f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty}[f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty}[f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$
- $\lim_{n\to\infty}[f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Properties

- **Transitivity**

$$f(n) = \Theta(g(n)) \,\&\, g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \,\&\, g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \,\&\, g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o\,(g(n)) \,\&\, g(n) = o\,(h(n)) \Rightarrow f(n) = o\,(h(n))$$
$$f(n) = \omega(g(n)) \,\&\, g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

- **Reflexivity**

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Properties

- **Symmetry**

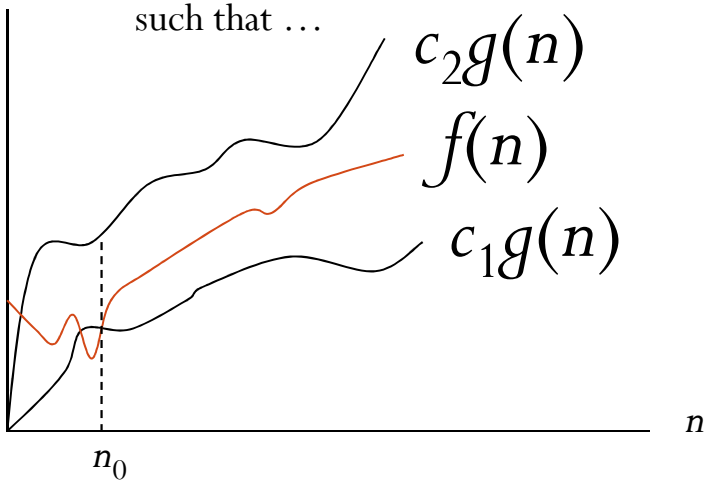
$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

- **Complementarity**

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

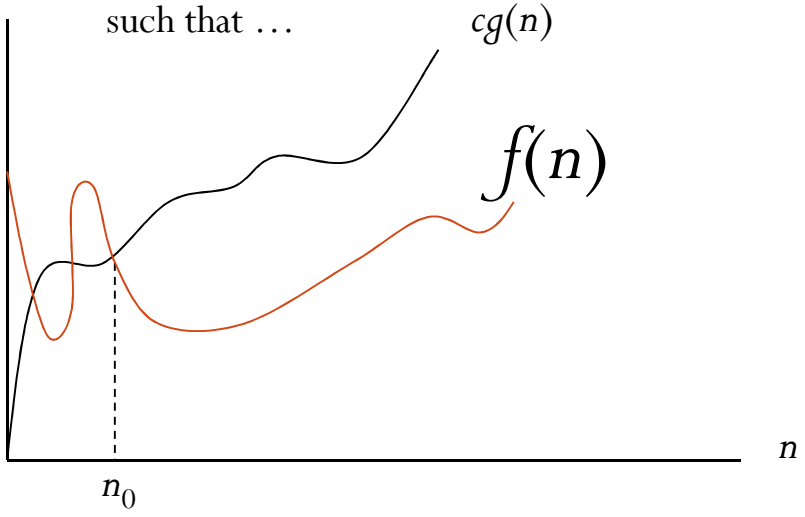
$$f(n) = o(g(n)) \text{ iff } g(n) = \omega((f(n))$$

There exist positive constants c_1 and c_2
such that there is a positive constant n_0
such that …



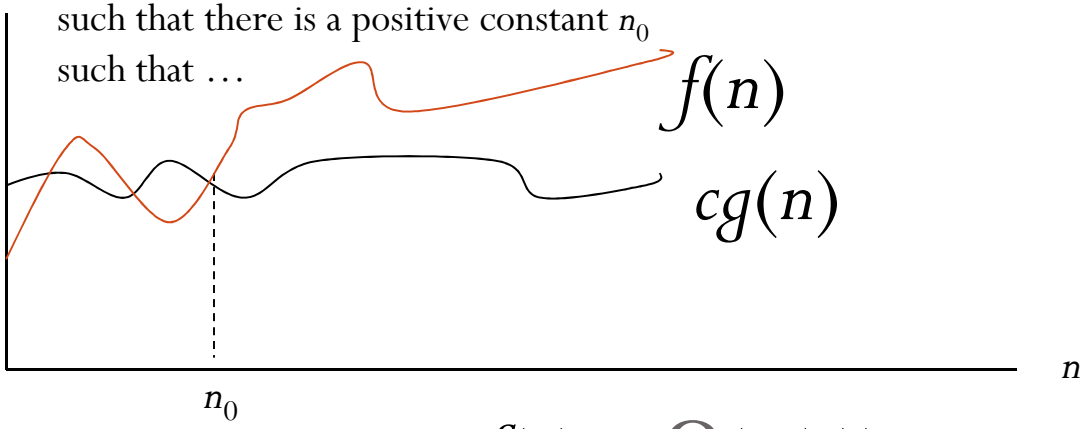
$$f(n) = \Theta(g(n))$$

There exist positive constants c
such that there is a positive constant n_0
such that …



$$f(n) = O(g(n))$$

There exist positive constants c
such that there is a positive constant n_0
such that …



$$f(n) = \Omega(g(n))$$

$f(n) = O(g(n))$	There exist $c > 0$ and $n_0 > 0$ such that: $0 \leq f(n) \leq cg(n)$ for each $n \geq n_0$
$f(n) = \Omega(g(n))$	There exist $c > 0$ and $n_0 > 0$ such that: $0 \leq cg(n) \leq f(n)$ for each $n \geq n_0$
$f(n) = \Theta(g(n))$	There exist $c_1, c_2 > 0$ and $n_0 > 0$ such that: $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for each $n \geq n_0$

Exercise

- Explain why the statement, "The running time of algorithm A is at least $O(n^2)$" is meaningless.

- **Solution:** Let the running time be $T(n)$.
- $T(n) \geq O(n^2)$ means that $T(n) \geq f(n)$ for some function $f(n)$ in the set $O(n^2)$.
- This statement holds for any running time $T(n)$, since the function $g(n) = 0$ for all n in $O(n^2)$, and running times are always nonnegative.

  Thus, the statement tells us nothing about the running time.

Shortcomings of asymptotic analysis

- let algorithm ***A*** be asymptotically better than algorithm ***B.*** Here are some common issues with algorithms that have better asymptotic behavior:

Implementation complexity

- Algorithms with better complexity are often (much) more complicated. This can increase coding time and the constants.

Small input sizes

- Asymptotic analysis ignores small input sizes. At small input sizes, constant factors or low order terms could dominate running time, causing *B* to outperform *A*.

Worst case versus average performance

- If *A* has better worst case performance than *B*, but the average performance of *B* given the expected input is better, then *B* could be a better choice than *A*. Conversely, if the worst case performance of *B* is unacceptable (say, for life-threatening or

What is worst case efficiency?

The worst case efficiency of an algorithm is its efficiency for the worst case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size.

What is best case efficiency?

The Best case efficiency of an algorithm is its efficiency for the best case input of size n which is an input of size n for which the algorithm runs fastest among all possible inputs of that size.

What is average case efficiency?

The average case efficiency of an algorithm is its efficiency for the random input of size n which makes some assumption about possible inputs of size n.

- **The 0/1 KNAPSACK Problem** : The Problem deals with choosing a subset of n objects with weights wi and profits pi (i ranges from 1 to n) such that the net profits of the selected objects is maximum while the knapsack capacity (say m) is not exceeded.

Assignment

- Suppose a certain algorithm has been empirically shown to sort 100,000 integers or 100,000 floating-point numbers in two seconds. Would we expect the same algorithm to sort 100,000 strings in two seconds? Explain.

Assignment

- Suppose you need to search a given ordered array of n integers for a specific value. The entries are sorted in non-decreasing order.
  1. Give a simple algorithm that has a worst-case complexity of $\Theta(n)$
  2. We can do better by using the "binary search" strategy. Explain what it is and give an algorithm based on it. What is the worst-case complexity of this algorithm?

- Use limits to prove the following:

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

Problems

- Explain the meaning of asymptotic bounds. Explain the importance of asymptotic notation. Determine if an assertion involving asymptotic bounds is valid. Ex: Is $3n^2+3n = O(n)$?
- Compare the growth rates of two expressions.

  Ex: $2^{\{\lg n\}}$ vs $(\lg n)^2$.

- Explain why the statement, "The running time of algorithm A is at least $O(n^2)$" is meaningless.