Time complexity and Space Complexity Analysis

Definition

- An algorithm is a finite sequence of step by step, discrete, unambiguous instructions for solving a particular problem
  - has input data, and is expected to produce output data
  - each instruction can be carried out in a finite amount of time in a deterministic way

Definition

- In simple terms, an algorithm is a series of instructions to solve a problem (complete a task)
- Problems can be in any form

Definition

- An algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of inputs in finite amount of time.
- **Criteria:**
  - **Input**
  - **Output**
  - **Definiteness**
  - **Effectiveness**
  - **Finiteness**

Properties of Algorithm

1. Non ambiguity:
   - Each instruction should be clear and precise.
2. Range of Input:
   - Input range should be specified
   - It should be finite
3. Multiplicity:
   - Same algorithm can be represented in several ways.
4. Speed:
   - Algorithm should be efficient
   - Should produce the output with fast speed
5. Finiteness:
   - Algorithm should be finite
   - it should terminate

Analysis of Algorithm

- Analyzing the algorithm means understanding the specification of the algorithm and conclude some useful information about its implementation.

  - Determine the running time of program(Time complexity)
  - Determine the space required for program data(Space Complexity)
  - Simplicity
  - Generality
  - Range of inputs

Performance Analysis

- Performance analysis is a process of measuring time and space required by a correct program for valid set of inputs

Performance Analysis

- Two criteria are used to judge algorithms:

  (i) time complexity

  (ii) space complexity.

- Space Complexity of an algorithm is the amount of memory it needs to run to completion.

- Time Complexity of an algorithm is the amount of CPU time it needs to run to completion.

Space Complexity

- Memory space S(P) needed by a program P, consists of two components:
  - A fixed part: needed for instruction space (byte code), simple variable space, constants space etc. → c
  - A variable part: dependent on a particular instance of input and output data. → S_p(instance)
- $S(P) = c + S_p(\text{instance})$

Space Complexity: Example 1

```
Algorithm abc (a, b, c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

For every instance 3 computer words required to store variables: a, b, and c. Therefore $S_p()= 3$. $S(P) = 3$.

Space Complexity: Example 2

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

Space Complexity: Example 2.

- Every instance needs to store array `a[]` & `n`.
  - Space needed to store `n` = 1 word.
  - Space needed to store `a[ ]` = n floating point words (or at least n words)
  - Space needed to store `i` and `s` = 2 words

- $S_p(n) = (n + 3)$. Hence $S(P) = (n + 3)$.

Time Complexity

- Time required T(P) to run a program P also consists of two components:
  - A fixed part: compile time which is independent of the problem instance → c.
  - A variable part: run time which depends on the problem instance → t_p(instance)
- $T(P) = c + t_p(\text{instance})$

Time Complexity

- How to measure T(P)?
  - Measure experimentally, using a "stop watch"

    → T(P) obtained in secs, msecs.
  - Count program steps → T(P) obtained as a <u>step count.</u>
- Fixed part is usually ignored; only the variable part $t_p()$ is measured.

Time Complexity

- What is a <u>program step</u>?

  - a+b+b*c+(a+b)/(a-b) → one step;

  - comments → zero steps;

  - `while (<expr>) do` → step count equal to the number of times <expr> is executed.

  - `for i=<expr> to <expr1> do` → step count equal to number of times <expr1> is checked.

Time Complexity: Example 1

	Statements	S/E	Freq.	Total
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

$\boxed{2n+3}$

Time Complexity: Example 2

	Statements	S/E	Freq.	Total
1	Algorithm Sum(a[], n, m)	0	–	0
2	{	0	–	0
3	for i=1 to n do;	1	n+1	n+1
4	for j=1 to m do	1	n(m+1)	n(m+1)
5	s = s+a[i][j];	1	nm	nm
6	return s;	1	1	1
7	}	0	–	0

$\boxed{2nm+2n+2}$

Analysis of Algorithms

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

The Execution Time of Algorithms

- Each operation in an algorithm (or a program) has a cost.

  ➔ Each operation takes a certain amount of time.

`count = count + 1;` ➔ take a certain amount of time, but it is constant

A sequence of operations:

```
count = count + 1;
sum = sum + count;
```

Cost: c_1

Cost: c_2

➔ Total Cost $= c_1 + c_2$

The Execution Time of Algorithms (cont.)

Example: Simple If-Statement

	Cost	**Times**
`if (n < 0)`	c1	1
`absval = -n`	c2	1
`else`		
`absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

The Execution Time of Algorithms (cont.)

Example: Simple Loop

	Cost	**Times**
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

Total Cost = $c1 + c2 + (n+1)*c3 + n*c4 + n*c5$

➔ The time required for this algorithm is proportional to n

The Execution Time of Algorithms (cont.)

Example: Nested Loop

Code	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to n^2

General Rules for Estimation

- **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- **Nested Loops**: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.
- **Consecutive Statements:** Just add the running times of those consecutive statements.
- **If/Else**: Never more than the running time of the test plus the larger of running times of S1 and S2.

Some Mathematical Facts

- Some mathematical equalities are:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + \ldots + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$$

Best Case

- If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case time complexity.

Worst Case

- If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called worst case time complexity.

Average Case

- The time complexity that we get for certain set of inputs is average, then for corresponding input such a time complexity is called average case.

Order of Growth

- Measuring the performance of an algorithm in relation with the input size n is called order of growth.

n	**log n**	**n log n**	**n^2**	**2^n**
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4,294,967,296

Growth of Functions

- To choose the best algorithm, we need to check efficiency of each algorithm.

- The efficiency can be measured by computing time complexity of each algorithm.

- Asymptotic notation is a shorthand way to represent the time complexity

Asymptotic Analysis

- To compare two algorithms with running times $f(n)$ and $g(n)$, we need a **rough measure** that characterizes **how fast each function grows.**
- <u>*Hint:*</u> use *rate of growth*
- Compare functions in the limit, that is, **asymptotically!**

  (i.e., for large values of n)

Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)

- The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Asymptotic Notation

- O notation: asymptotic "less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality":
  - f(n)= Θ (g(n)) implies: f(n) "=" g(n)

Big-O Notation

- Big-oh is the formal method of expressing the upper bound of an algorithm's running time.
- It is the measure of longest amount of time it could possibly take for the algorithm to complete.
- More formally,

  For non negative functions, f(n) and g(n),if there exists an integer n_o and a constant c>o such that for all integers n> $n_{o.}$

  $$f(n) <= cg(n).$$

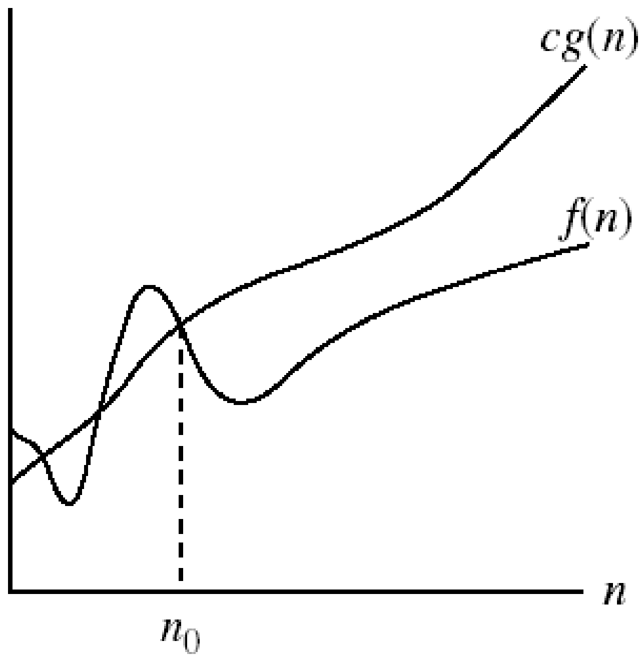
- Then f(n) is big-oh of g(n).

  This is denoted as "f(n) ϵ O(g(n))"

Asymptotic notations

- *O-notation*

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



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or O (n)
  It is, at most, roughly *proportional* to n.
- $f_B(n)=n^2+1$ is *order* n^2, or $O(n^2)$. It is, at most, roughly proportional to n^2.
- In general, any $O(n^2)$ function is faster- growing than any $O(n)$ function.

Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or O (n)
  It is, at most, roughly *proportional* to n.
- $f_B(n)=n^2+1$ is *order* n^2, or $O(n^2)$. It is, at most, roughly proportional to n^2.
- In general, any $O(n^2)$ function is faster- growing than any $O(n)$ function.

Examples

- $2n^2 = O(n^3)$:

$$2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1 \text{ and } n_0 = 2$$

- $n^2 = O(n^2)$: $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1 \text{ and } n_0 = 1$

- $1000n^2+1000n = O(n^2)$:

$$1000n^2+1000n \leq 1000n^2+ n^2 =1001n^2 \Rightarrow c=1001 \text{ and } n_0 = 1000$$

- $n = O(n^2)$:

$$n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1 \text{ and } n_0 = 1$$

No Uniqueness

- There is no unique set of values for n_0 and c in proving the asymptotic bounds
- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \le 100n + n = 101n \le 101n^2$

    for all $n \ge 5$

    $n_0 = 5$ and $c = 101$ is a solution
  - $100n + 5 \le 100n + 5n = 105n \le 105n^2$

    for all $n \ge 1$

    $n_0 = 1$ and $c = 105$ is also a solution

Must find **SOME** constants c and n_o that satisfy the asymptotic notation relation

Big-Omega Notation

- For non-negative functions, f(n) and g(n), if there exists an integer n_o and a constant c>o such that for all integers n> n_o,f(n)>=cg(n) then f(n) is big omega of g(n).
- This is denoted as

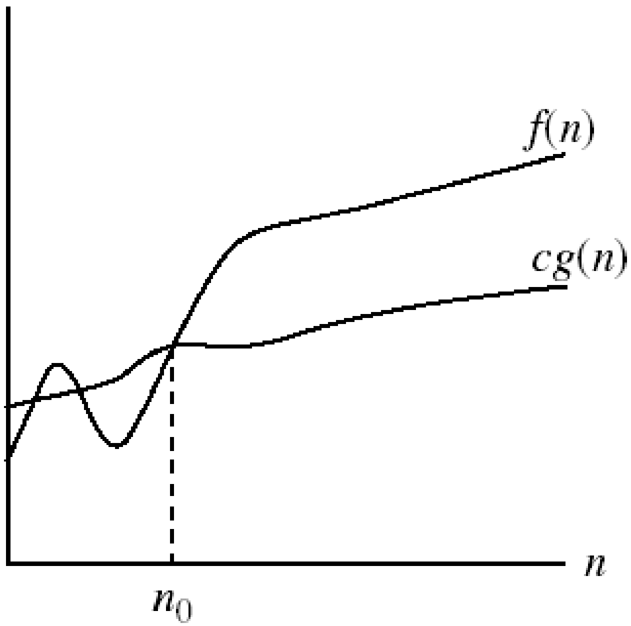
  "f(n) ϵ Ω(g(n))"
- G(n) is a lower bound function.

- It describes the best that can happen for a given data size.

Big –Omega Notation

- Ω - *notation*

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



$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Examples

- $5n^2 = \Omega(n)$

  $\exists\, c, n_0$ such that: $0 \le cn \le 5n^2 \Rightarrow cn \le 5n^2 \Rightarrow c = 1$ and $n_0 = 1$

- $100n + 5 \ne \Omega(n^2)$

  $\exists\, c, n_0$ such that: $0 \le cn^2 \le 100n + 5$

  $100n + 5 \le 100n + 5n \ (\forall\, n \ge 1) = 105n$

  $cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

  Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

  $\Rightarrow$ contradiction: n cannot be smaller than a constant

- $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$

Big-Theta Notation

- For non negative functions, f(n) and g(n),if there exists an integer n_0 and a constant $c_1$and c_2i.e., c_1>oand c_2>o such that for all integers n> n_0.

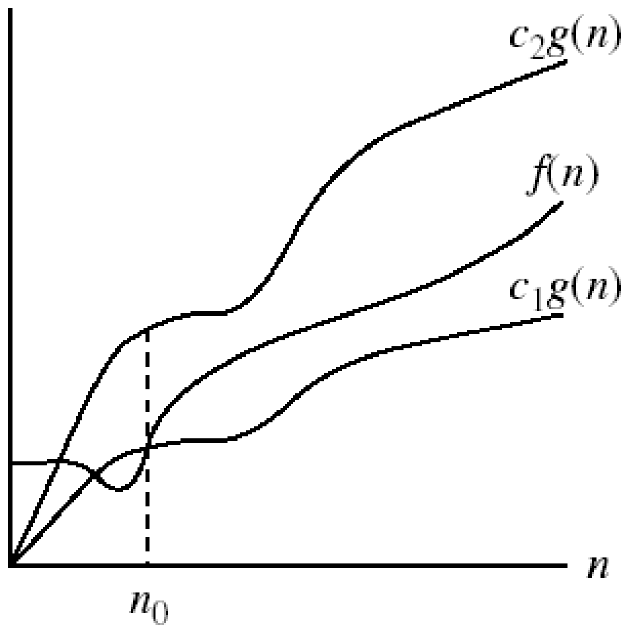
  $c_1 g(n) <= f(n) <= c_2 g(n)$.

- Then f(n) is big-oh of g(n).

- This is denoted as "$f(n) \epsilon \Theta(g(n))$"

Asymptotic notations (cont.)

$\Theta(g(n)) = \{f(n) :$ 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 $n \ge n_0\}$.



Θ(g(n)) is the set of functions with the same order of growth as g(n)

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Examples

- $n^2/2 - n/2 = \Theta(n^2)$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \le \frac{1}{2} n^2 \;\forall n \ge 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \ge \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \;(\forall n \ge 2) = \frac{1}{4} n^2 \quad \Rightarrow \quad c_1 = \frac{1}{4}$
- $n \ne \Theta(n^2)$: $c_1 n^2 \le n \le c_2 n^2$

$\Rightarrow$ only holds for: $n \le 1/c_1$

Examples

$6n^3 \neq \Theta(n^2)$: $c_1 n^2 \leq 6n^3 \leq c_2 n^2$

$\Rightarrow$ only holds for: $n \leq c_2 / 6$

- $n \neq \Theta(\log n)$: $c_1 \log n \leq n \leq c_2 \log n$

$\Rightarrow c_2 \geq n/\log n, \forall\, n \geq n_0$ – impossible

Common orders of magnitude

Table 1.4 Execution times for algorithms with the given time complexities

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.005 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 μs = 10^{-6} second.
†1 ms = 10^{-3} second.

Common orders of magnitude