Data Structures

Course Overview

Instructor

Dr. Yasser Abdelhamid

Course Objectives

- To identify the different data structures including lists, linked lists, stacks, queues, trees, graphs and hash tables.
- To learn how the choice of data structure affects program performance.
- To study the relevant algorithms needed to handle different types of data structures.
- To learn how to estimate the complexity of different algorithms.
- To specify and implement various useful abstract data types (ADTs).

Grading

- **50 marks for final exam**
- **20 marks for midterm exam**
- **30 for coursework (quizzes)**

Textbooks

- http://people.cs.vt.edu/~shaffer/Book/

The most recent version is Edition 3.2.0.10, dated March 28, 2013.

What is an algorithm ?

- Definition (from **Wikipedia**)

" ...a finite **set of operations** for accomplishing some task, **given an initial state**, will **terminate** in a corresponding recognizable **end-state**"

What is an algorithm ?

- With reference to **Computer programs,** an algorithm is a **sequence of instructions** that solves a given problem
  - **Operates on data**
  - **Receives input values**
  - **Generates output values**
  - **Terminate after finite number of steps**

What is an algorithm ?

The term "**algorithm**" comes from **Al-Khwarizmi**, a Persian mathematician of the IX century.

Why algorithms are important?

- **Practical Reasons**

We need efficient algorithms for solving most practical problems.

- **Technological Reasons**

Algorithms can be implemented as programs on computers

Example of Algorithms

- **Sorting a set of objects...**

Example of Algorithms

- Finding an item in a set of objects...

Example of Algorithms

- Finding the shortest path between two destinations ...

Data Structures

- **Programs = Algorithms + Data structures**

Data Structure is mainly concerned with finding the best representation or organization of data In the memory that leads to efficient processing

Data Structures

- **Programs = Algorithms + Data structures**

Choice of data structure can affect the efficiency of an algorithm

e.g., accessing an element in a list.

Analysis of an Algorithm

Analysis of an algorithm

- It is extremely important to determine the expected **resources** required by an algorithm.
  - Execution Time
  - Amount of Memory
  - Other resources (disk usage, communication ports, software resources, etc.)

Analysis of an algorithm

- Parameters affecting the analysis are:
  - The size of the input.
  - Input values
  - Other (execution model.........)

Algorithm performance

- Of primary consideration when estimating an algorithm’s performance is the number of **basic operations** required by the algorithm to process an input of a certain size.
- Size is often the number of inputs processed.
- A basic operation must have the property that its time to complete does not depend on the particular values of its operands.
- Adding or comparing two integer variables are examples of basic operations in most programming languages.

Example 1

- The size of the problem is A.length
- The basic operation is to compare an integer's value to that of the largest value seen so far.
- It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array.

```
/** @return Position of largest value in  array A */
static int largest(int[] A) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<A.length; i++)   // For each element
    if (A[currlarge] < A[i])       // if A[i] is larger
       currlarge = i;         //   remember its position
  return currlarge;           // Return largest position
}
```

Example 1

- Let n be the number of elements of the array.
- Let c be a constant time that is required to make a comparison operation which is our basic operation in this algorithm.
- Let T be the time required to run the algorithm, and we refer to this time by the function T(n).
- We will always assume T(n) is a non-negative value.
- The total time to run the algorithm is therefore approximately c*n, i.e. cn.

$$T(n) = cn$$

Example 2

- The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value.
- Let this constant time be $c1$.
- then $T(n) = c1$

Example3: Comparing Algorithm Efficiency

- Consider the following three Algorithms for computing $1+2+\ldots+n$, $n > 0$

Algorithm A	
sum = 0 for i = 1 to n sum = sum +i	

Example: Comparing Algorithm Efficiency

- Consider the following three Algorithms for computing 1+2+…+n , n > 0

Algorithm A	Algorithm B	

sum = 0
for i =1 to n
        sum = sum +i
	sum = 0
	for i = 1 to n
	  {for j =n to i
	        sum = sum +1 }	

Example: Comparing Algorithm Efficiency

- Consider the following three Algorithms for computing 1+2+…+n , n > 0

Algorithm A	Algorithm B	Algorithm C

sum = 0
for i =1 to n
    sum = sum +i
	sum = 0
	for i = 1 to n
	 {for j =n to i
	    sum = sum +1 }
		sum = n * (n +1)/ 2

Example: Comparing Algorithm Efficiency

- Consider the following three Algorithms for computing 1+2+…+n , n > 0

Algorithm A	Algorithm B	Algorithm C

sum = 0
for i =1 to n
    sum = sum +i
	sum = 0
	for i = 1 to n
	 {for j =n to i
	    sum = sum +1 }
		sum = n * (n +1)/ 2

- The number of operations required in each algorithm is

	Alg. A	Alg. B	Alg. C
Assignments	n +1	1 + n (n +1)/2	1
Additions	n	n (n+1)/2	1
Multiplications			1
Divisions			1
Total	2 n +1	n^2+ n +1	4

Complexity analysis: Definitions

Complexity = cost in terms of

- **Execution time T(n) "Time Complexity"** refers to the amount of time needed to solve a problem instance
- **Required Storage S(n) "Space Complexity"** refers to the amount of memory needed to solve a problem instance

n is the size of the input

Example: For a sorting algorithm

n is the number of elements in the set

Complexity analysis: Assumptions

- **Complexity analysis** Must be **independent** of the type of computer.

- **Asymptotic complexity**

We are interested in values of $\mathbf{n} \to \infty$

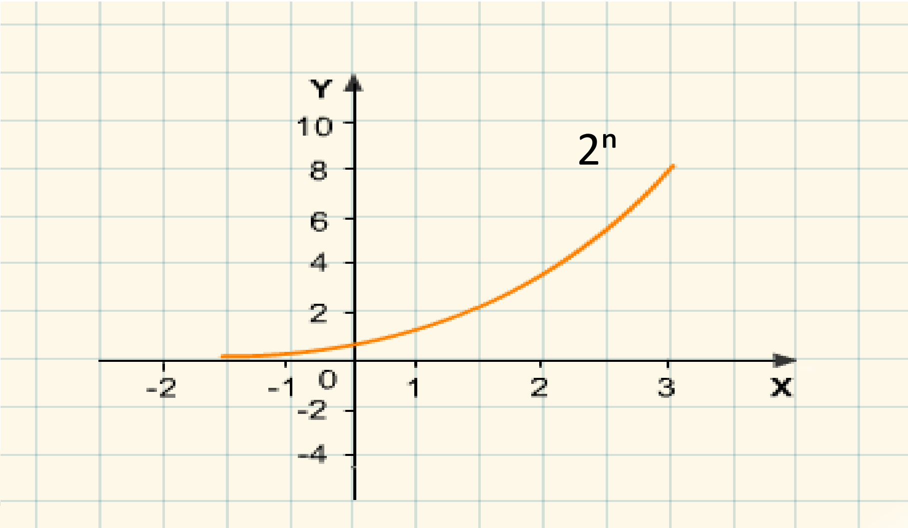
Rate of Growth

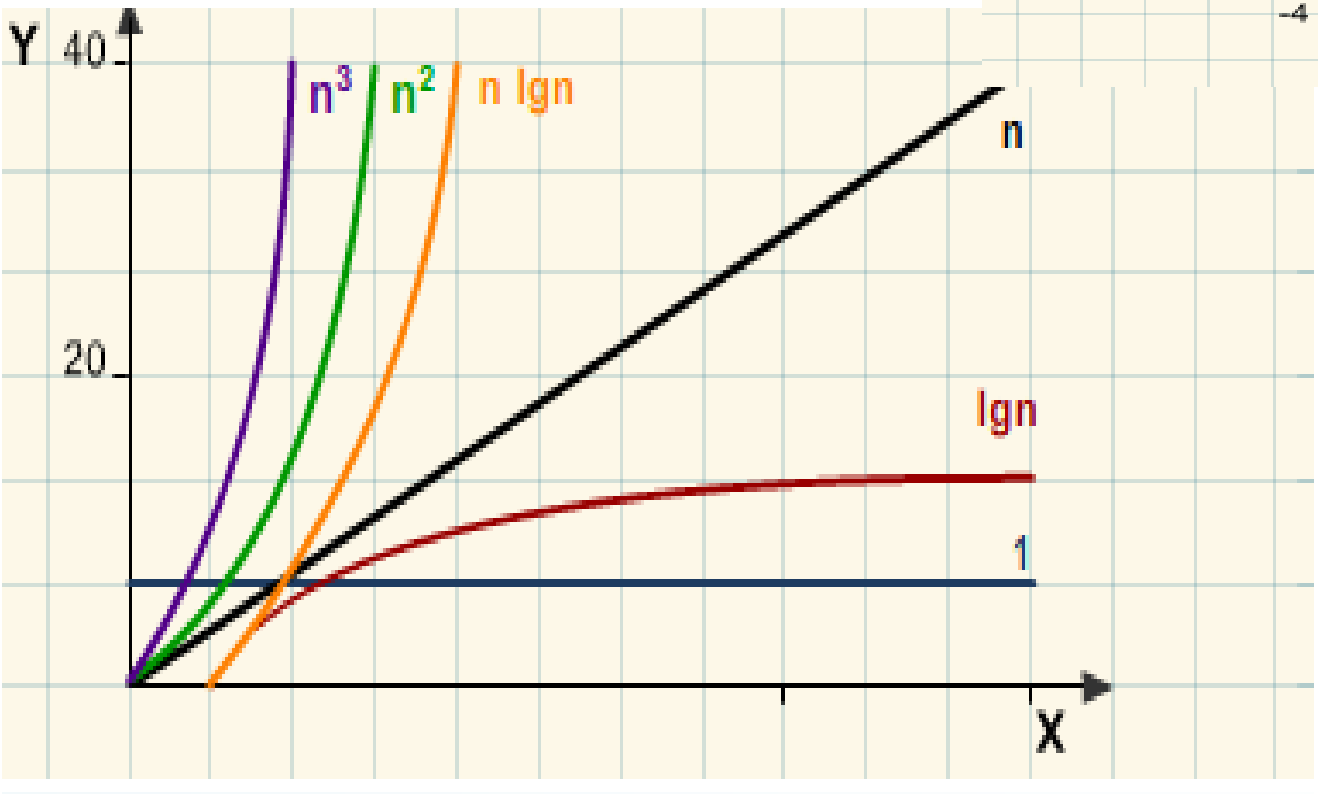
Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
log n	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in a unsorted array
nlog n	Linear Logarithmic	Sorting n items by 'Divide and Conquer'
n^2	Quadratic	Shortest path between 2 nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Rate of Growth

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log n)**	**O(n)**	**O(n log n)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Seven functions used in analysis of algorithms





Complexity analysis and algorithms

- Evaluation of the efficiency of algorithms is essential !

Not Just algorithms ...

But efficient algorithms ...

Asymptotic notation

Asymptotic notation

- Three standard notations: **O,** Ω**,** θ

O, Ω provides a loose bound (upper and lower respectively)

θ Provides a tight bound.

- **O,** Ω**,** θ identify classes of functions

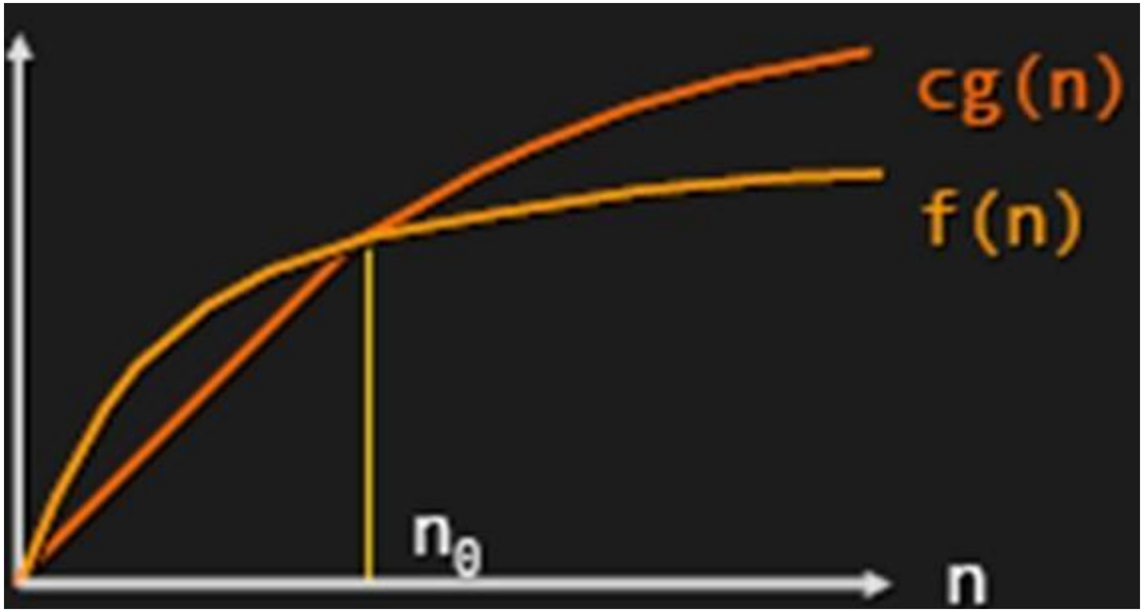
Notation: **T(n)=f(n)= O(g(n))**

O Notation ("Big-O")

Definition

$f(n)=O(g(n))$ if $c > 0$ and $n_0 \geq 0$ such that

$f(n) \leq c.g(n) \ \forall \ n \geq n_0$



O Notation ("Big-O")

- The **O** notation does not always provide a tight bound

Example: $f(n)=O(n^2)$

After some value of n, f(n) is bounded by $\mathbf{n^2}$ but f(n) is bounded also by $O(\mathbf{n^3})$.

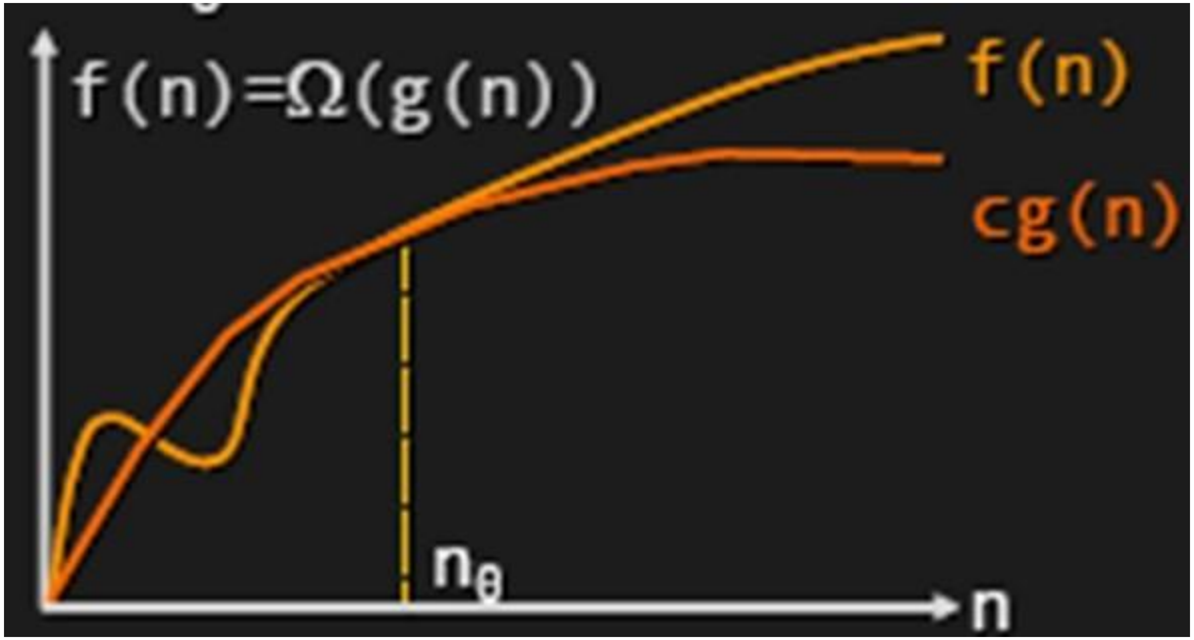
Objective: is to find the tightest upper bound

Ω Notation ("Big-omega")

Definition

$f(n)=\Omega(g(n))$ if $c > 0$ and $n_0 \geq 0$ such that

$f(n) \geq c.g(n) \ \forall \ n \geq n_0$

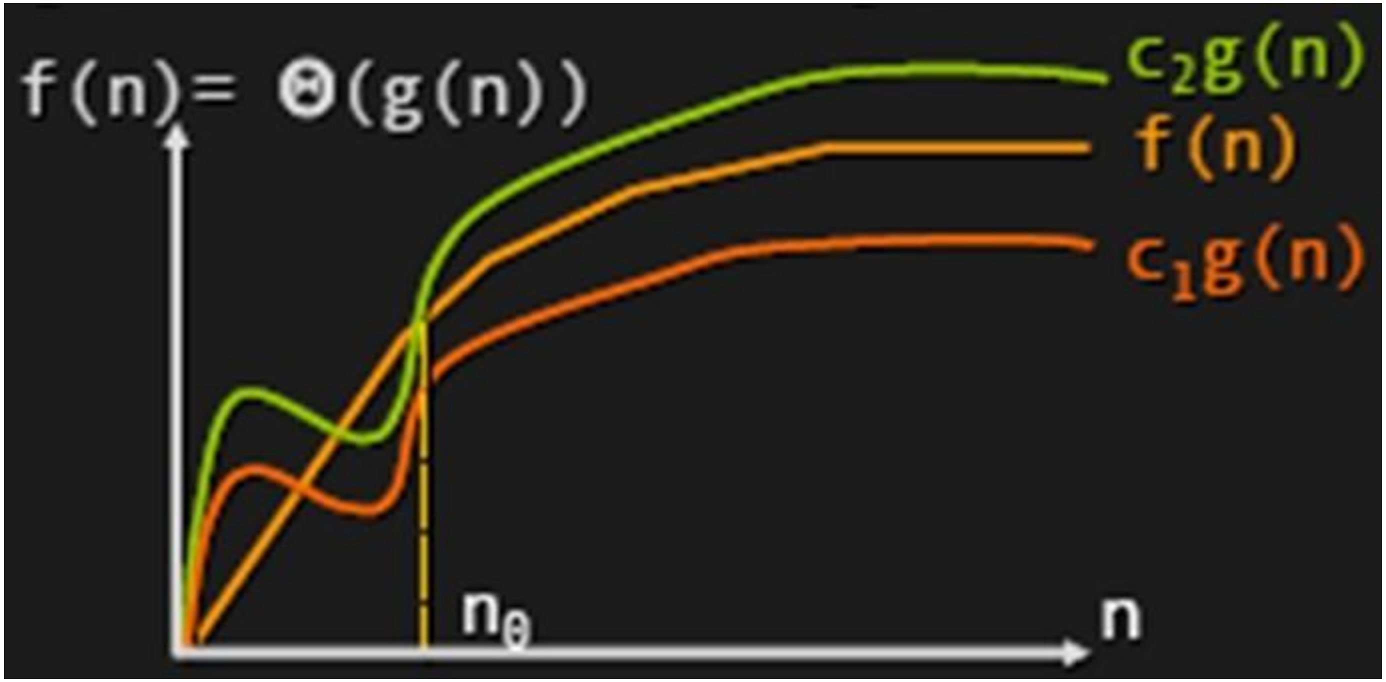


θ Notation ("Big-Theta")

Definition

$f(n)=\theta(g(n))$ if c_1, $c_2 > 0$ and $n_0 \geq 0$ such that

$c_1.g(n) \leq f(n) \leq c_2.g(n)\ \forall\ n \geq n_0$



θ Notation ("Big-Theta")

- $f(n)=\theta(g(n))$ then it must hold that

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

And

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

Asymptotic notation: summary

- O provides a **loose upper** bound
- Ω provides a **loose lower** bound
- θ provides a **tight** bound

O, Ω, θ Notations: **Examples**

- Example 1:

Prove that T(n)=3n+2 is an element of O(**n**)

solution

To get T(n) = O(n), we need to verify that **T(n) $\leq$ c.n**

At c=4 and n_0= 2. **T(n)=3n+2 $\leq$ 4.n** for $n_0 \geq 2$

O, Ω, θ Notations: **Examples**

- <u>Example 2:</u>

Prove that $T(n)=3n+2$ is an element of $\Omega(\mathbf{n})$

<u>solution</u>

To get $T(n) = \Omega(n)$, we need to verify that $\mathbf{T(n) \geq c.n}$

At $c=3$ and $n_0= 1$. $\mathbf{T(n)=3n+2 \geq 3.n}$ for n_0.

O, Ω, θ Notations: **Examples**

- Example 3:

Prove that $T(n)=3n+2$ is an element of $\theta(\mathbf{n})$

solution

To get $T(n) = \theta(n)$, we need to verify that

$c_1.n \leq \mathbf{T(n)} \leq c_2.n$

At $c_1=3$, $c_2=4$, $n_0 \geq 2$,

$$\mathbf{3n} \leq \mathbf{3n+2} \leq \mathbf{4n} \text{ for } \forall\ n_0 \geq 2.$$

O, Ω, θ Notations: **Examples**

- Example 4: $T(n) = 10n^2+4n+2$, **$T(n) = O(n^2)$?**

Solution, **YES**

To get $T(n) = O(n^2)$, we need to verify that

$T(n) \leq c.n^2$

At $c=11$ and $n_0= 5$. **$T(n)= 10n^2+4n+2 \leq 11n$** for
$n_0 \geq 5$

O, Ω, θ Notations: **Examples**

- Example 5: $T(n) = 10n^2+4n+2$, **$T(n) = \Omega(n^2)$?**

Solution, **YES**

To get $T(n) = \Omega(n^2)$, we need to verify that

$T(n) \geq c.n^2$

At $c=1$ and $n_0= 1$. **$T(n)= 10n^2+4n+2 \geq 1n$** for $\forall n_0$

O, Ω, θ Notations: **Examples**

- Example 6: $T(n) = 10n^2+4n+2$, **$T(n) = \theta(n^2)$?**

Solution, **YES**

To get $T(n) = \theta(n^2)$, we need to verify that

$c_1.n^2 \le \mathbf{T(n)} \le c_2.n^2$

At $c_1=5$, $c_2=12$, $n_0 =2$,

$\mathbf{5n^2} \le \mathbf{10n^2+4n+2} \le \mathbf{12n^2}$ for $\forall\ n_0 \ge 2$.

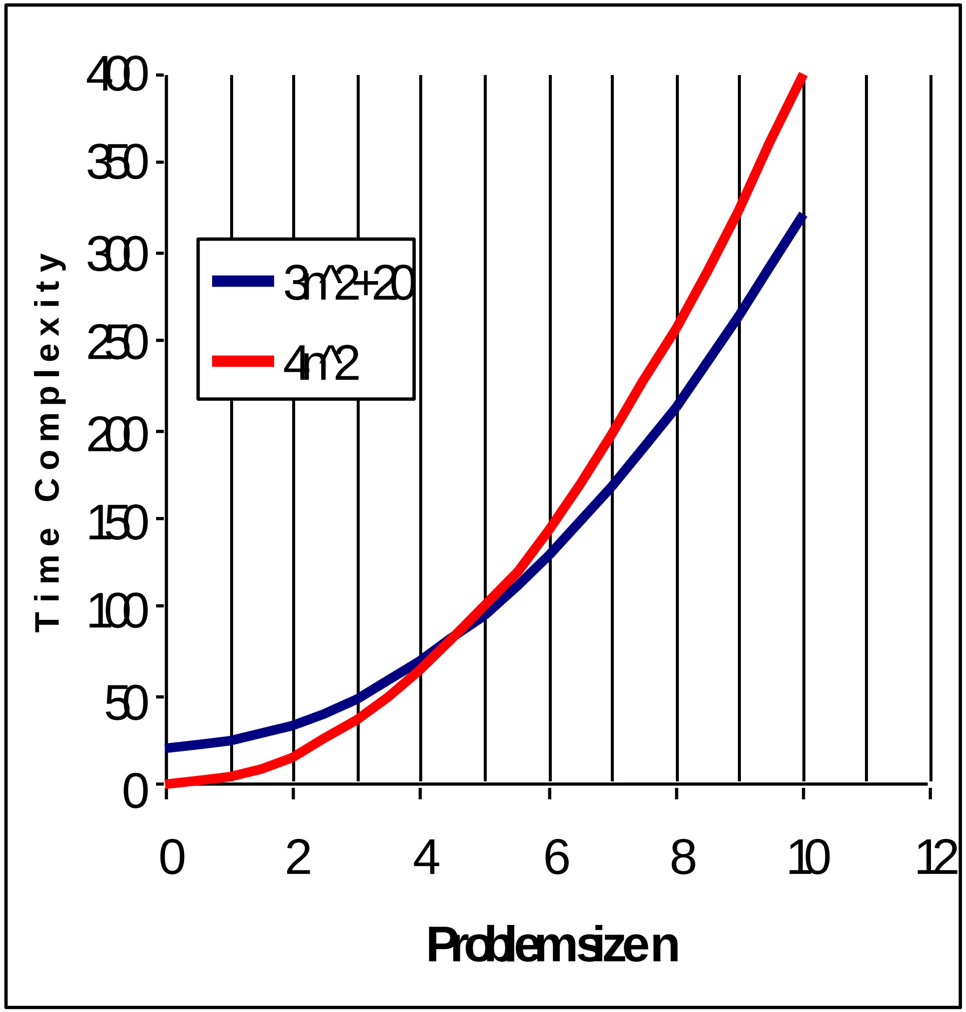
O, Ω, θ Notations: **Examples**

Example 7:

- Show that **$f(n)=3n^2+20$** has **$O(n^2)$**
  - We need to find two real numbers $n_0>0$ and $c>0$ where the inequality $0\leq 3n^2+20 \leq cn^2$ is fulfilled
  - Let $n_0=5$ and $c=4$

    ➔ $0\leq 3n^2+20 \leq 4n^2$

    ➔ $3n^2+20 \in O(n^2)$



O, Ω, θ Notations: **Examples**

Example 8:

- What is the Big-Oh of multiplying two arrays of size n?

$$T(n) = 1 + (n+1) + n + 1$$

$$= 2n + 3$$

$$= O(n)$$

```
Algorithm multiply (x[], y[], n)
      sum ← 0;
      for (i=0; i<n; i++)
          sum ← sum +x[i]*y[i];
      return sum;
```

Properties of Asymptotic notations

Properties of Asymptotic notations

- **Transitivity**

$f(n)=\theta(g(n))$ and $g(n) = \theta(h(n))$

Then

$$f(n) = \theta(h(n))$$

Properties of Asymptotic notations

- **Transitivity**

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

Then

$$f(n) = O(h(n))$$

Properties of Asymptotic notations

- **Transitivity**

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

Then

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

Properties of Asymptotic notations

- **Symmetry**

  $f(n) = \theta(g(n)) \iff g(n) = \theta(f(n))$

- **Transpose Symmetry**

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

Properties of Asymptotic notations

- **Reflexivity**

$$f(n) = \theta(f(n))$$

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

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

Properties of Asymptotic notations

- **Sum and Maximum**

$$f_1(n)+f_2(n)+ \ldots +f_m(n)$$

$$\Downarrow$$

$$\theta(\max(f_1(n)+f_2(n)+ \ldots +f_m(n)))$$

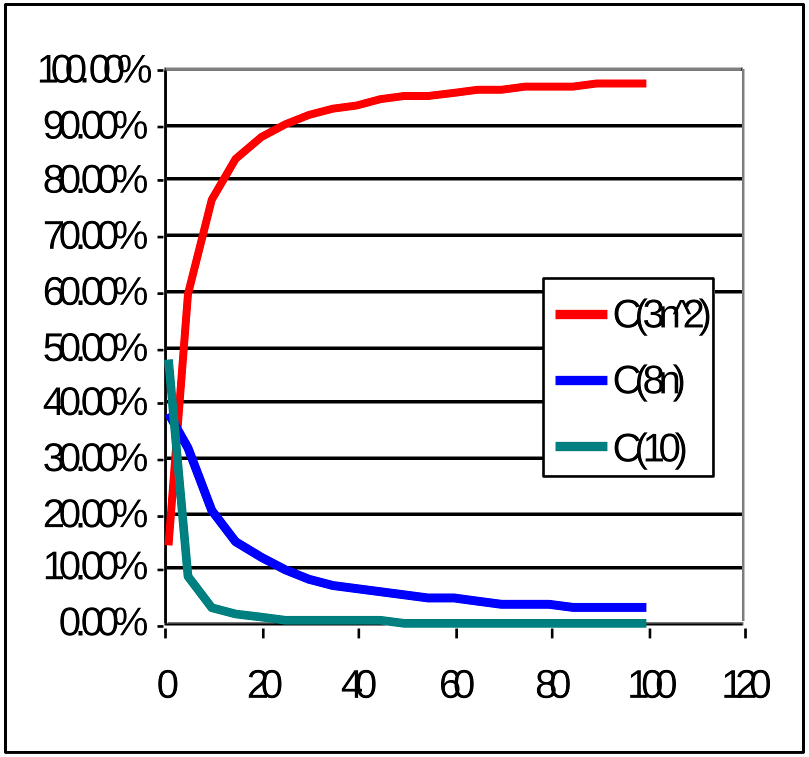
Example:

$3n^2+n+7 = \theta(3n^2) = \theta(n^2)$

Example

- Assume the actual time complexity of an algorithm is $T(n) = 3n^2+8n+10$, what is the approximate time complexity of that algorithm?
  - Since $T(n)$ is getting bigger (i.e. monotonically increasing) by increasing the problem size n, we can study the contribution of each term; $3n^2$, $8n$, and 10, on the increase of $T(n)$

Experiment#1



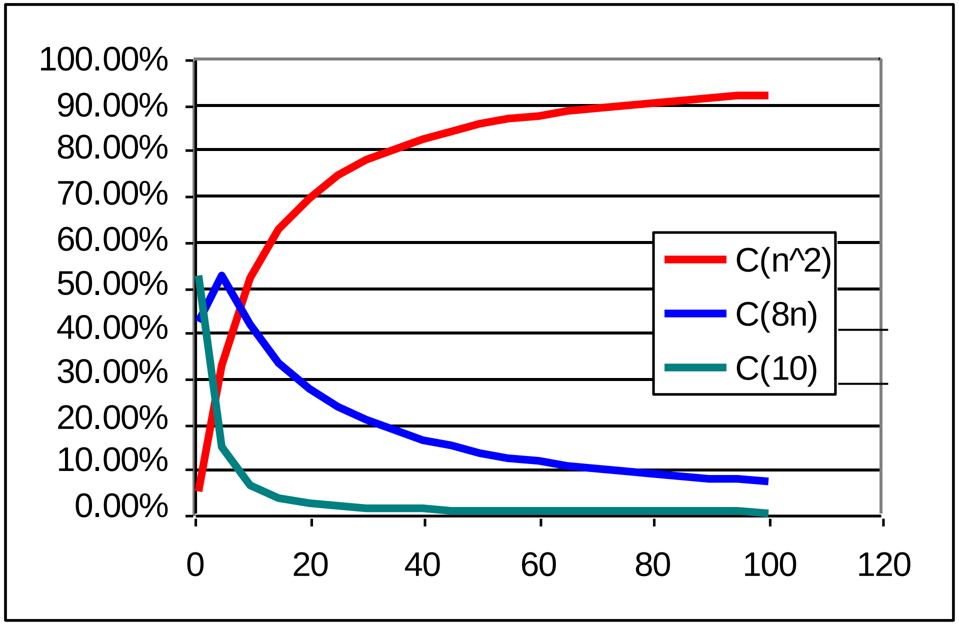
n	*3*n^2*	*8*n*	*10*	*T(n)*	*C(3n^2)*	*C(8n)*	*C(10)*
1	3	8	10	21	14.29%	38.10%	47.62%
5	75	40	10	126	59.52%	31.75%	8.73%
10	300	80	10	392	76.53%	20.41%	3.06%
15	675	120	10	808	83.54%	14.85%	1.61%
20	1200	160	10	1374	87.34%	11.64%	1.02%
25	1875	200	10	2090	89.71%	9.57%	0.72%
30	2700	240	10	2956	91.34%	8.12%	0.54%
35	3675	280	10	3972	92.52%	7.05%	0.43%
40	4800	320	10	5138	93.42%	6.23%	0.35%
45	6075	360	10	6454	94.13%	5.58%	0.29%
50	7500	400	10	7920	94.70%	5.05%	0.25%
55	9075	440	10	9536	95.17%	4.61%	0.22%
60	10800	480	10	11302	95.56%	4.25%	0.19%
65	12675	520	10	13218	95.89%	3.93%	0.17%
70	14700	560	10	15284	96.18%	3.66%	0.16%
75	16875	600	10	17500	96.43%	3.43%	0.14%
80	19200	640	10	19866	96.65%	3.22%	0.13%
85	21675	680	10	22382	96.84%	3.04%	0.12%
90	24300	720	10	25048	97.01%	2.87%	0.11%
95	27075	760	10	27864	97.17%	2.73%	0.10%
100	30000	800	10	30830	97.31%	2.59%	0.10%

As problem size n increases, the contribution of $3n^2$ term increases and other terms decrease!

Experiment#1

- **Observation**:
  - As **$n \rightarrow \infty$** the term **$3n^2$** dominates (i.e. approaches **100%**) while the other terms decease (i.e. approaches **0%**)
- **Conclusion**:
  - We can ignore the lower degree terms from the complexity function as **$n \rightarrow \infty$**
  - This leads to the first approximation of the previous complexity function to **$T(n) \approx 3n^2$**
- Now how about the coefficient **3**?

Experiment#2



If we ignore the coefficient of the highest degree term, it still dominates the other two terms as **n** is getting bigger

n	*C(n^2)*	*C(8n)*	*C(10)*
1	5.26%	42.11%	52.63%
5	32.89%	52.63%	14.47%
10	52.08%	41.67%	6.25%
15	62.85%	33.52%	3.63%
20	69.69%	27.87%	2.44%
25	74.40%	23.81%	1.79%
30	77.85%	20.76%	1.38%
35	80.49%	18.40%	1.12%
40	82.56%	16.51%	0.93%
45	84.23%	14.98%	0.79%
50	85.62%	13.70%	0.68%
55	86.78%	12.62%	0.60%
60	87.76%	11.70%	0.54%
65	88.61%	10.91%	0.48%
70	89.35%	10.21%	0.44%
75	90.00%	9.60%	0.40%
80	90.57%	9.06%	0.37%
85	91.09%	8.57%	0.34%
90	91.55%	8.14%	0.32%
95	91.96%	7.74%	0.30%
100	92.34%	7.39%	0.28%

Experiment#2

- **<u>Observation:</u>**
  - Ignoring the coefficient of the highest degree term does not affect the contribution of that term on the growth of the complexity function **T(n)**, i.e. it still dominates the other two terms as long as **$n \rightarrow \infty$**
- **<u>Conclusion:</u>**
  - As **$n \rightarrow \infty$** we can simply drop the coefficient of the highest degree term since it is still dominating the other terms in the complexity function and therefore **$T(n) \approx n^2$**

Example

- It is required to run **3** independent algorithms A1, A2, and A3 simultaneously in a parallel system with 3 processors, what is the overall complexity when A1 has $\mathrm{O}(9\mathrm{n}^3 + 4)$, A2 has $\mathrm{O}(10\mathrm{n}^2 + 3\mathrm{n} + 2)$, and A3 has $\mathrm{O}(2\mathrm{n}^4 + \mathrm{n} - 1)$?

Solution

- The overall complexity will biased to the algorithm **A3** which has **largest** complexity.

i.e., T(n)= $O(2n^4 + n - 1) = \mathbf{O(n^4)}$

Example - Find the time complexity?

```
Sum=0;
for (i = 1; i < n; i++)
{
   sum = sum +i;
 }
System.out.println(sum);
```

1

n – 1 + 1 = n

n – 1

1

Then, T(n)=1 + n + n -1 + 1 = 2n + 1 = O(n)

Example - Find the time complexity?

```
for (i = 1; i < n; i++)
{
  P = i;
  Min = A[P];
  for (j = i+1; j <= n; j++)
  {
          if (A[j] < Min)
          {
             P = j;
            Min = A[P];
          }
          A[P] = A[i];
          A[i] = Min;
   }
 }
```

Example - Find the time complexity?

```
for (i = 1; i < n; i++)
{
   P = i;
   Min = A[P];

   for (j = i+1; j <n+1; j++)
   {
         if (A[j] < Min)
         {
            P = j;
            Min = A[P];
         }
   }
}
```

n – 1 + 1 = n

n – 1

n – 1

$$\sum_{i=1}^{n-1} n + 1 - (i + 1) + 1 = A$$

A-1

A-1

A-1

- $A = \sum_{i=1}^{n-1} n + 1 - (i + 1) + 1$

- $A = \sum_{i=1}^{n-1} n - i + 1 = n + (n - 1) + \cdots + 3 + 2$

- $A = n + (n - 1) + \cdots + 3 + 2 = \frac{(n-1)}{2}[2 * 2 - (n - 1 - 1) * 1] = \frac{(n-1)(n+2)}{2}$

- $A = \frac{(n-1)(n+2)}{2} = \frac{n^2-4n-2}{2}$

- **The Overall Complexity = T(n)**

- $\mathrm{T(n)} = n + (n-1) + (n-1) + A + (A-1) + (A-1) + (A-1)$
- $\mathrm{T(n)} = 3n - 2 + 4A - 3 = 3n + 4A - 5$
- $\mathrm{T(n)} = 3n + 4 * \frac{(n^2-4n-2)}{2} - 5$
- $\mathrm{T(n)} = 3n + 2n^2 - 8n - 4 - 5$
- $\mathrm{T(n)} = 2n^2 - 5n - 9 \longrightarrow \mathrm{T(n)} = O(n^2)$

Series & Summations

Some useful series

- **Arithmetic series**

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

- **Geometric series**

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \quad (x \neq 1)$$

- **Special case (at x < 1)**

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

Some useful series

- **Harmonic series**

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

- **Other useful series**

$$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \ldots + n^p \approx \frac{1}{p+1} n^{p+1}$$