Lecture 1
Introduction

Algorithm

- A well-defined computational procedure that takes some value (or set of values) as *input* and produces some value (or set of values) as *output*
- Provides a step by step method for solving a computational problem.
- Is not dependent on any particular programming language, machine or compiler.
- A general computational procedure that solves a well-defined specific problem.

Criteria

- Input – there are zero or more quantities which are externally supplied.
- Output – at least one quantity is produced
- Definiteness – each instruction must be clear and unambiguous
- Finiteness – it terminates after a finite number of steps
- Effectiveness – every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper.

Computational Problem (Example)

- Input:

  A set of *n* real numbers $(a_1, a_2, \ldots, a_n)$

- Output:

  A value *S* where

  $$S = g + l$$
  $$g = max\ (a_1, a_2, \ldots, a_n)$$
  $$l = min\ (a_1, a_2, \ldots, a_n)$$

The sum of the greatest number and lowest number.

Algorithm (Example)

Let A[i] be the i^{th} number on the list $(a_1, a_2, \ldots a_n)$

1 Max, min = A[1]

2 For $i = 2$ to n

3 If $A[i] > max$ then $max = A[i]$

4 ElseIf $A[i] < min$ then $min = A[i]$

5 Next i

6 Return $max + min$

Design Issues in Algorithms

- Correctness – does the algorithm solve the computational problem?

- Efficiency – how fast can the algorithm run?

Consider Another Scenario

- Problem: Robot Tour Optimization
- Input: A set of S of n points in the plane
- Output: What is the shortest cycle tour that visits each point in the set S?

Assumptions: robot moves with fixed speed, thus the travel time between 2 points are proportional to their distance

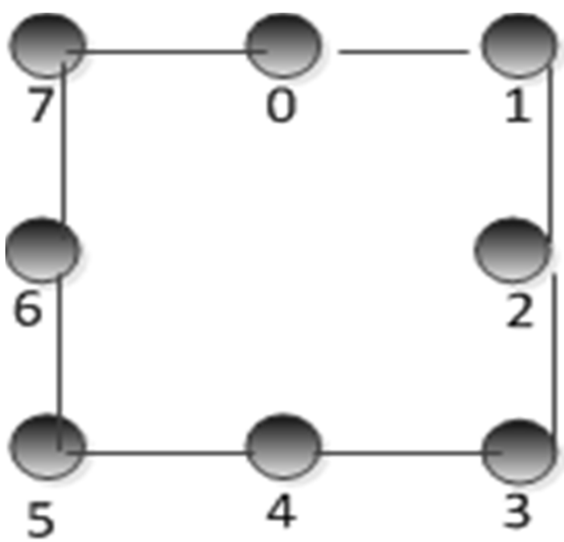
Possible Solution

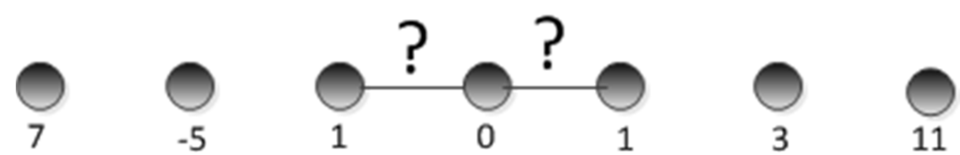
- Nearest-neighbor heuristic
  - Starting from some point p_o, walk first to the nearest neighbor p_1.
  - From p_1, walk to its nearest neighbor unvisited neighbor, excluding p_o
  - Repeat the process until we run out of unvisited points
  - Return to p_o to close off the tour.
- Pseudocode
  - Pick & visit an initial point p_o from P
  - $p = p_o$
  - $I = 0$
  - While there are still unvisited points
    - $I = I + 1$
    - Select p_i to be the closes unvisited point to p_{i-1}
    - Visit p_i
  - Return to p_o from p_{n-1}

Design Issues in Algorithms

- Correctness – does the algorithm solve the computational problem?

- Efficiency – how fast can the algorithm run?





Analyzing Algorithms, why?

- Intended use of the algorithm
- predicting the resources that the algorithm requires, such that evaluation of its suitability for various applications can be done
  - memory
  - communication bandwidth
  - computer hardware
  - computational time

Why is there a need for algorithm analysis?

- To study its behavior
  - What happens if the input size is increased?
- To predict its performance
  - Time (processing speed)
  - Space (memory)
- Given two algorithms, A1 and A2 solving the same problem: which is better?
- Or given an existing algorithm, can a modified optimal algorithm be defined?

Assumption

- We are using a generic processor, Random Access Memory (RAM) model of computation
  - Instructions are executed ONE AT A TIME, no concurrent operations
- Algorithms implemented as computer programs

Algorithm Efficiency Analysis Methodologies

- A priori analysis – determines the efficiency of an algorithm based on or derived from mathematical or logical facts
- A posteriori analysis – determines the efficiency of an algorithm based on actual experiments

A Posteriori Analysis (Example)

	10	100
Algo 1	1 sec	10 sec
Algo 2	3 sec	15 sec

Algo 1 seems more efficient than Algo2

	10	100	1000
Algo 1	1 sec	10 sec	100 sec
Algo 2	3 sec	15 sec	30 sec

Is Algo1 still faster than Algo2?

A Priori Analysis (Example)

Assumptions:

- *Instructions are executed sequentially*
- *Each instruction takes c time units*

Let A[i] be the i^{th} number on the list $(a_1, a_2, \ldots a_n)$

1 Max, min = A[1] *c*

2 For *i* = *2* to *n*

3 If *A[i]* > *max* then *max* = *A[i]*

4 ElseIf *A[i]* < *min* then *min* = *A[i]*

5 Next *i*

6 Return *max* + *min* *c*

[n-1] [2 c (worst-case comparison) + c (assign)]

Total = (3n – 1) *c*

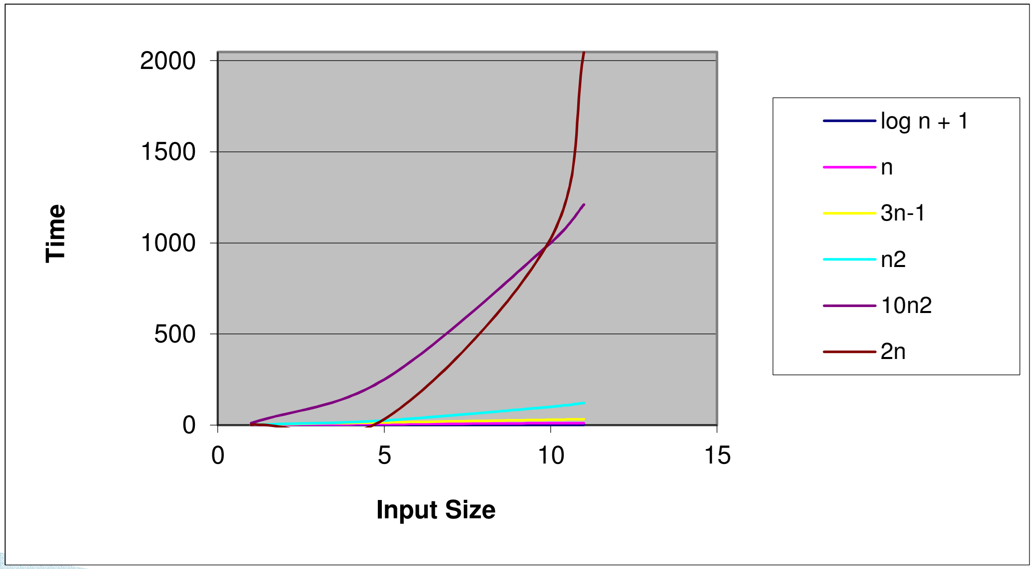
Analyzing Algorithms

- Time taken by an algorithm grows with the size of the input
  - *input size*
    - Number of items in the input
    - Number of bits needed
  - *running time*
    - Number of primitive operations
    - Number of “steps” executed
- Our concern: *Rate Of Growth* or *Order Of Growth*

Growth of Functions

	1	5	10	11
(log n) +1	1	1.7	2	2.04
n	1	5	10	11
3n-1	2	14	29	32
n^2	1	25	100	121
$10n^2$	10	250	1000	1210
2^n	2	32	1024	2048

In time-complexity analysis it is important to note how fast the algorithm performs over the size of the input and other factors

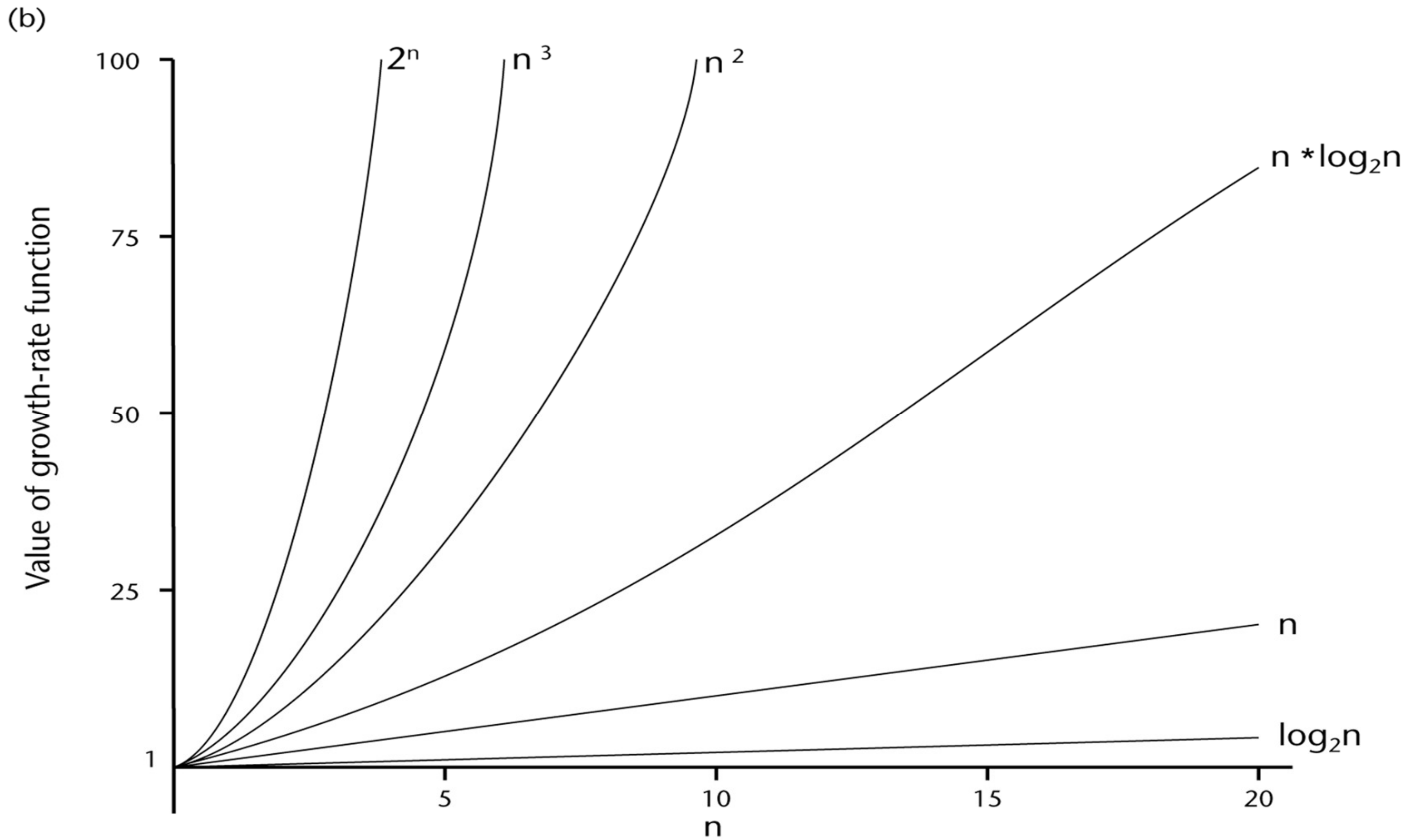


A Comparison of Growth-Rate Functions

(a)

Function	n: 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A Comparison of Growth-Rate Functions (cont.)



Growth-Rate Functions

- If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?
- If its order is:

O(1) ➔ $T(n) = 1$ second

O($\log_2 n$) ➔ $T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds

O(n) ➔ $T(n) = (1*16) / 8 = 2$ seconds

O($n*\log_2 n$) ➔ $T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds

O(n^2) ➔ $T(n) = (1*16^2) / 8^2 = 4$ seconds

O(n^3) ➔ $T(n) = (1*16^3) / 8^3 = 8$ seconds

O(2^n) ➔ $T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds $= 256$ seconds

Program = $t(n) = 60n^2 + 5n + 1$

n	$t(n)$	$60n^2$
10	6,051	6,000
100	600,501	60,000
1000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

$t(n)$ grows "like" $60n^2$

Assuming $t(n) = 60n^2 + 5n + 1$ is measured in terms of seconds.

So in terms of minutes: $n^2 + 5n/60 + 1/60$

Observations on Growth

- The dominant term (term with the fastest growth rate) in the function determines the behavior of the algorithm
- Any exponential function of n dominates any polynomial function of n
- A polynomial degree k dominates a polynomial of degree m iff $k > m$
- Any polynomial function of n dominates any logarithmic function of n
- Any logarithmic function of n dominates a constant term

Order of Growth

- The order of growth is a function of the dominant term of the running time
- The dominant term is the term that contributes the most significant increase in $T(n)$ as n increases
- The coefficient of the dominant term is ignored

Exercise

What is the growth rate corresponding to the following running time?

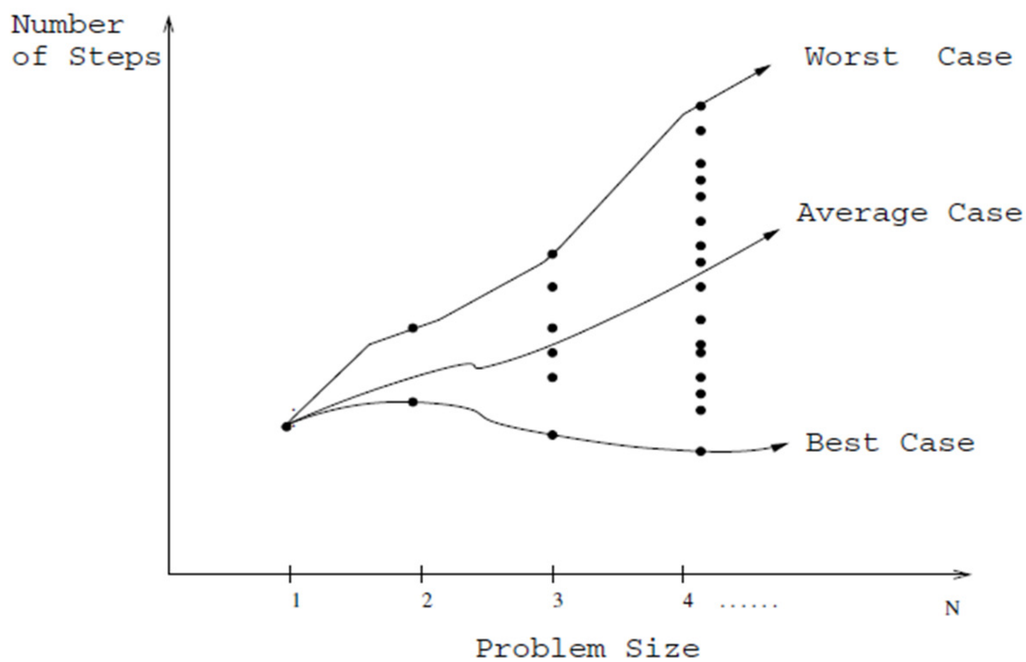
1. $3n + 5n - 2$
2. $6n^2 + 7n + 3$
3. $9n^3 + 6n^2 + n + 2$

Asymptotic Bounds – Big-Oh, Theta, and Omega

- It is hard to get the exact running-time of an algorithm
- Asymptotic Bounds are used instead to describe the complexity of the algorithms
- Asymptotic Bounds describes only the growth rates of the algorithm as the input size approaches infinity and ignoring most of the small inputs and constant factors
- Among these bounds are: Big-Oh, Big-Omega and Theta

For Simplicity...

- The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken in any instance of size n.
- The best-case complexity of the algorithm is the function defined by the minimum number of steps taken in any instance of size n.
- The average-case complexity of the algorithm, which is the function defined by the average number of steps over all instances of size n.



Asymptotic Bounds – Big-Oh, Theta, and Omega

- The Big-Oh of a function g(n) is O(f(n)), iff there exist a positive number c and n_0 such that:

  $$0 <= g(n) <= c\ f(n) \text{ where } n >= n_0$$

- Describes an *asymptotically loose upper bound* of the algorithm
- Represents the *worst-case running time* of the algorithm

Example

- g(n) = $2n^2 + 3$
- The big-Oh of $g(n)$ is $O(n^2)$
- Proof:

$$2n^2 + 3 <= c\, n^2$$

Divide both sides by n^2

$$2 + 3/n^2 <= c$$

$g(n) <= c\, n^2$, for $c = 5$, $n_0 = 1$, $n >= n_0$

Properties of Growth-Rate Functions

1. *We can ignore low-order terms in an algorithm's growth-rate function.*
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.

2. *We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.*
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

3. *$O(f(n)) + O(g(n)) = O(f(n)+g(n))$*
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 +4n^2)$ ➔ So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

Asymptotic Bounds – Big-Oh, Theta, and Omega

- The Big-Omega of a function g(n) is $\Omega(f(n))$, iff there exist a positive number c and n_0 such that:

  $$0 <= c\ f(n) <= g(n) \text{ where } n > n_0$$

- Describes an *asymptotically loose lower bound* of the algorithm
- Represents the *best-case running time* of the algorithm

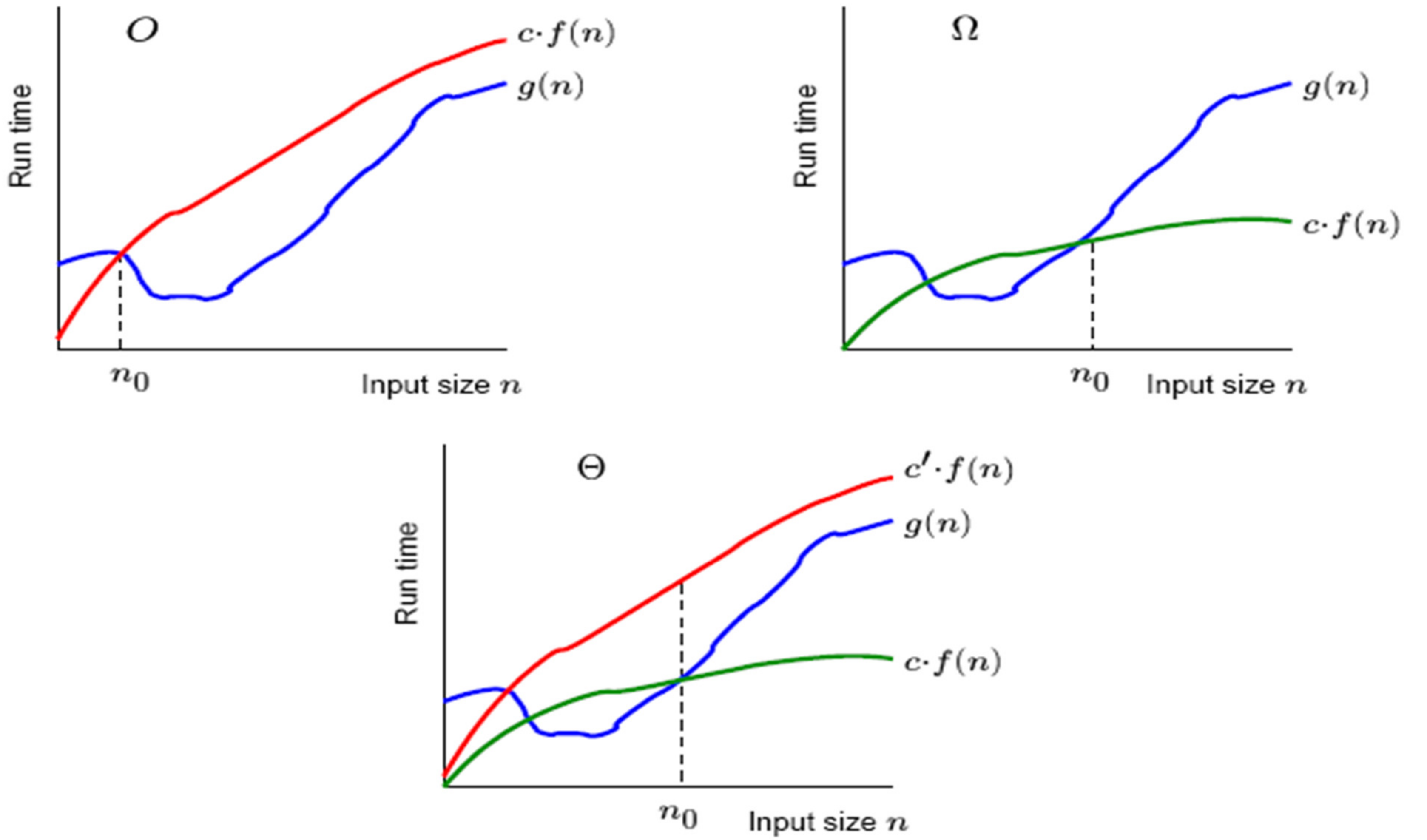
Asymptotic Bounds – Big-Oh, Theta, and Omega

- The Theta of a function g(n) is θ(f(n)), iff there exist a positive number c, c' and n_0 such that:

$$c_1 f(n) <= g(n) <= c_2 f(n) \text{ where } n > n_0$$

- Describes an *asymptotically tight bound* of the algorithm
- Represents the *average-case running time* of the algorithm

Asymptotic Bounds – Big-Oh, Theta, and Omega



Common Upper Bounds

$O(1)$	constant
$O(\log n)$	logarithm
$O(n)$	linear
$O(n \log n)$	linear-log
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

NP Complete Problem

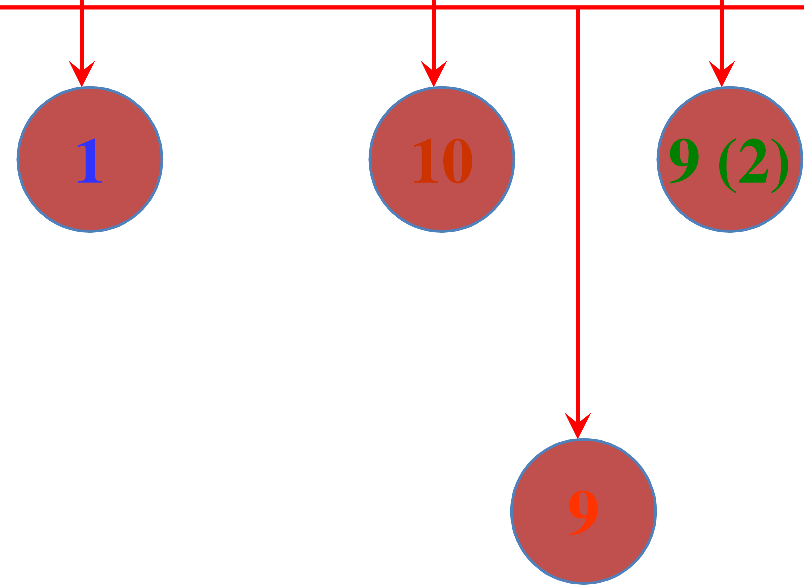
Operation Count

- Language: C Language
- Declarations
  - none with no initializations
  - count of 1 with initializations
- Delimiters (such as { and })
  - none
- Function Heading
  - none
- Operators (Arithmetic, Relational, Logical)
  - for simplicity, each operator has a count of 1
- Expressions
  - sum of all operators

- Assignment Statement
  - 1count for assignment operator
  - 2 counts for ++, - -, +=, -=, *=, /=, %=
- Function call
  - 1(function call) + operation count for the operators + operation count of the function
- if or if-else Statement
  - operation count of conditional statement + Maximum Operation Count (if_block, else_block)
- for Statement
- while Statement
- do-while Statement
- Nested Loops

Analyze the following code

```
for (j = 1; j < 10; j++)
  printf ("I love Ice Cream");
```



$$\therefore \mathbf{T}(n) = 38$$

Analyze the following code

```
for (j = 1; j <= 10; j++)
    printf ("I love Ice Cream");
```

1

11

10(2)

10

$$\therefore \mathbf{T}(n) = 42$$

Analyze the following code

```
for (j = 5; j < 10; j++)
    printf ("I love Ice Cream");
```

1

6

5(2)

5

$$\therefore\ \mathbf{T}(n) = 22$$

Analyze the following code

```
for (j = 5; j <= 10; j++)
    printf ("I love Ice Cream");
```

1 7 6(2)

6

$$\therefore \mathbf{T}(n) = 26$$

Let's Generalize

```
for (j = 1; j < 10; j++)
    printf ("I love Ice Cream");
```

$1 + 10 + 9(2)$

9

```
for (j = 5; j < 10; j++)
    printf ("I love Ice Cream");
```

$1 + 6 + 5(2)$

5

Let’s Generalize (with respect to frequency count)

```
for (j = a; j < b;  j++)
    printf ("I love Ice Cream");
```

1   b – a + 1   b – a

b – a

Let’s Generalize

```
for (j = 1; j <= 10; j++)
    printf (“I love Ice Cream”);
```

$1 + 11 + 10(2)$

10

```
for (j = 5; j <= 10; j++)
    printf (“I love Ice Cream”);
```

$1 + 7 + 6(2)$

6

Let's Generalize (with respect to frequency count)

```
for (j = a; j <= b;    j++)
   printf ("I love Ice Cream");
```

1

b – a + 2

b – a + 1

b – a + 1

Sample Problems

```
(a) for (j = 0; j < n; j++)
     printf ("Sample Problem 1\n");

(b) for (j = 0; j < n; j++)
    {
       printf ("Operation Count - ");
       printf ("Sample Problem 2\n");
    }
```

(c)

```
int factorial (int nVal)
{
   int  j;
   int  nFactorial = 1;

   for (j = 1; j <= nVal; j++)
      nFactorial *= j;

   return nFactorial;
}
```

(d)

```
int factorial (int nVal)
{
    int  j;
    int  nFactorial = 1;

    for (j = nVal; j > 0; j--)
        nFactorial *= j;

    return nFactorial;
}
```

(e)

```
int n, i, j;
scanf ("%d", &n);
for (i = 0; i <  n; i++)
    for (j = 0;  j< n;  j++)
        printf ("%d", i * j);
```

Frequency Count

- Number of statements or steps needed by the algorithm to finish
- Simple statements:
  - $a \leftarrow 10$ Count: 1
  - $b \leftarrow a * 2$ Count: 1

Frequency Count

- Conditional Statements

```
if (<condition>)
    <S1>;
else
    <S2>;
```

Sum of the following:

- 1 + Max{ Count(<S1>), Count(<S2>) }

Frequency Count – Example

$$
\begin{array}{ll}
\text{if } x > 1 & \\
\quad y \leftarrow 10; & 1 \\
\text{else} & \\
\{ \quad y \leftarrow 20; & 1 \\
\quad\ z \leftarrow 30; & 1 \\
\} &
\end{array}
\quad 2
$$

Total = 3

Frequency Count

- Loop Statements

for i ← <lb> to <ub> ub - lb + 2
<S1> ub - lb + 1

Example

for i ← 1 to n n - 1 + 2 = n+1
x ← x + 1 n

Total = 2n+1

Frequency Count - Example

```
                                                   1
if x < 1
        y ← 10     1
else
        if x < 2                         1
        {       y ← 20;
                z ← 30        2                          2x+2
        }
         else
        {   for i ← 1 to x   x+1    = 2x+1
                 print(i)     x
        }
```

Total = 2x+3

Frequency Count – Exercise

1. $k \leftarrow 500;$

```
for i ← 1 to k-1
    z ← z + 1
```

2. for $k \leftarrow 0$ to n

```
{  print (k);
   print (n-k);
}
```

Frequency Count

- Nested Statements

$$\text{for } i \leftarrow 1 \text{ to } n \qquad n - 1 + 2 = n+1$$

$$(n) = \text{<S1>} \left[\begin{array}{ll} \text{for } j \leftarrow 1 \text{ to } n & (n+1)(n) \\ \quad x \leftarrow x + 1 & (n)(n) \end{array} \right.$$

$$\text{Total} = n+1 + n^2+n + n^2$$

$$= 2n^2 + 2n + 1$$

Frequency Count - Example

	for $i \leftarrow 2$ to $n-1$	$n-1 - 2 + 2 = n-1$
<S1>	for $j \leftarrow 1$ to n	$(n+1)(n-2)$
$= (n-2)$	$x \leftarrow x + 1$	$(n)(n-2)$

$$\text{Total} = n-1 + n^2-2n+n-2 + n^2-2n$$

$$= 2n^2 - 2n - 3$$

Frequency Count - Example

```
          for i ← 1 to n                 n+1
        [ {     x ← x + 1                n
<S1>              for j ← 3 to n+1            n(n)
= (n)             {  [ y ← y + 1      (n-1)(n)
          <S2>       [ z ← z + 1      (n-1)(n)
          = (n-1)    [
                  }
        [ }
```

$$\text{Total} = n+1 + n + n^2 + (n-1)n + (n-1)n$$

$$= 2n+1 + n^2 + n^2 - n + n^2 - n$$

$$= 3n^2 + 1$$

Frequency Count – Exercise

```
for i ← 1 to n
    for j ← 1 to n
        for k ← 1 to n
            z ← z + 1
```

Frequency Count

- Loop Statements

  do
  <S1>
  while <condition>

Ex.

What if while $x <= n$?
What if $x = 0$?

$x \leftarrow 1$ $=1$

do

$x \leftarrow x + 1$ $=n-x+1=n-2+1$ (exec at least once)

while $x < n$ $=n-x+1=n-2+1$ (test to stop loop)

Total = 2n-1

Frequency Count

- Loop Statements

  while <condition>

  <S1>

Ex.

$x \leftarrow 1$	1
while $x < n$	$n-x+1=n$
$x \leftarrow x + 1$	$n-1$
	Total = $2n$

What if while $x <= n$?

What if $x = 0$?

Frequency Count - Example

	Code	Count
1.	`x ← 1`	1
	`while x <= n`	n-x+2=n+1
	`    x ← x + 1`	n
		Total = 2n + 2
2.	`x ← 1`	1
	`do`	
	`    y ← y + 1`	n-1-x=n-2
	`    x ← x + 1`	n-1-x=n-2
	`while x <> n-1`	n-1-x=n-2
		Total = 3n - 5

Summation/Arithmetic Series

-arises in the analysis of iterative algorithms

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

$$\sum_{k=a}^{b} 1 = b - a + 1$$

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

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

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

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

Geometric Series:

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$$

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

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

Harmonic Series:

- arises in probabilistic analyses of algorithms

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$$

$$\sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$$

Telescoping series

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0$$

```
for (j = 3; j<= n; j++)
```

$1 \quad n-3+2 \quad 2(n-2)$

```
for (k = 0; k<= j; k++)
```

$\sum_{j=3}^{n} 1 \quad \sum_{j=3}^{n} (j+2) \quad \sum_{j=3}^{n} 2(j+1)$

```
printf("This is easy");
```

$\sum_{j=3}^{n} (j+1)$

for (j = 1; j < n; j++)

$1 \quad n-1+1 \quad 2(n-1)$

for (k = 2; k<= j + 1; k++)

$\sum_{j=1}^{n-1} 1 \quad \sum_{j=1}^{n-1} 2(j+1) \quad \sum_{j=1}^{n-1} 2(j)$

printf("This is easy");

$\sum_{j=1}^{n-1} (j)$

```
for (j = 3; j <= n; j++)
```

$1 \quad n-3+2 \quad 2(n-2)$

```
for (k = j; k<= n;  k++)
```

$\sum_{j=3}^{n} 1 \quad \sum_{j=3}^{n} (n-j+2) \quad \sum_{j=3}^{n} 2(n-j+1)$

```
printf("This is easy");
```

$\sum_{j=3}^{n} (n-j+1)$

for (j = 1; j < n; j++)

$1 \quad n-1+1 \quad 2(n-1)$

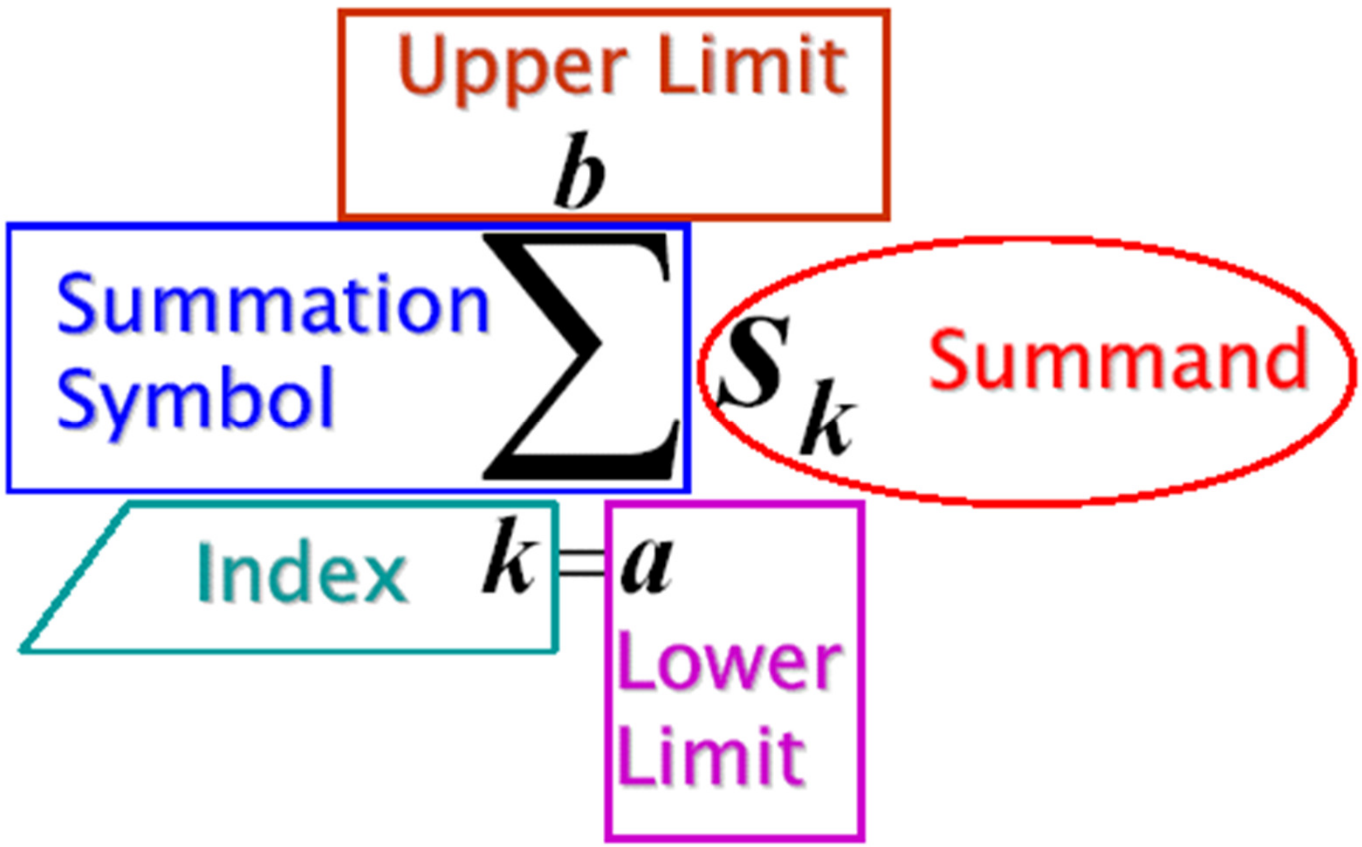
for (k = j+1; k<= 5; k++)

$$\sum_{j=1}^{n-1} 2 \qquad \sum_{j=1}^{n-1} (6-j) \qquad \sum_{j=1}^{n-1} 2(5-j)$$

printf("This is easy");

$$\sum_{j=1}^{n-1} (5-j)$$

Summation Notation



Using Summation

- for j = 1 to n

$$\sum_{j=1}^{n+1} 1 \begin{aligned} &= \langle ub \rangle - \langle lb \rangle + 1 \\ &= n+1 - 1 + 1 \\ &= n + 1 \end{aligned}$$

- for j = 1 to n−1

$$\sum_{j=1}^{n} 1 \begin{aligned} &= n - 1 + 1 \\ &= n \end{aligned}$$

- for j = 0 to n

$$\sum_{j=0}^{n+1} 1 \begin{aligned} &= n+1 - 0 + 1 \\ &= n + 2 \end{aligned}$$

Using Summation

- for j = 1 to n $\quad \sum_{j=1}^{n+1} 1$

  for k = 1 to n $\quad \sum_{k=1}^{n+1} \sum_{j=1}^{n} 1$

  x = x + 2; $\quad \sum_{k=1}^{n} \sum_{j=1}^{n} 1$

Nested Loops with Dependent Loop Control Variables

- for i = 1 to n
  for j = 1 to i
  x++;

- for i = 1 to n
  for j = i to n
  x++;

- for i = 1 to n
  for j = 1 to i
  for k = 1 to j
  x++;

Frequency Count – Exercise

1. for i ← 1 to n
    for j ← 1 to 2n
        x ← x + 1

2. for k ← 2 to n+1
    for j ← 3 to n-3
        x ← x + 1

Frequency Count – Exercise

3.
```
for i ← 2 to n+1
    for j ← 3 to n-3
        for k ← 4 to n-4
            x ← x + 1
```

4.
```
for i ← 1 to n
{   j ← 2
    while j<=n+3
    {   print(A[i],A[j-1]);
        j++;
    }
}
```

Frequency Count – Exercise

5. for i ← 1 to n
    for j ← n downto 1
        x ← x + 1

6. for i ← 1 to n-1
    for j ← 1 to i
        x ← x + 1

7. for i ← 4 to n
    for j ← 1 to i
        x ← x + 1

Frequency Count – Exercise

```
8. while (i < n)
   {   k = k+1;
       i = i+1;
   }

9. while (i>=n)
   {   k= k+1;
       x = x+1;
       i = i-1;
   }

10.  while (i > n)
   {     k = k+1;
         i = i-1;
   }

11.  while (b != n-10)
         b= b+1;

12.  do {  h = h-1;
     } while (h >= n);
```

Frequency Count - Exercise

```
13.  for i ← 1 to 2n
         for j ← 1 to 2i
             x ← x + 1

14.  for i ← 2 to n+2
         for j ← i to 2i
             x ← x + 1

15.  for i ← 1 to n
         for j ← 1 to i
             for k ← 1 to i
                 x ← x + 1
```

Summation - Exercise

$$\sum_{j=1}^{n} (3j^2 + n + 4)$$

$$\sum_{j=3}^{n} (j/4)^2$$

Recurrence

- An equation or inequality that describes a function in terms of its value on smaller inputs.
- Given a function defined by a recurrence relation, our objective is to determine a closed form of the function.
- Arises in the analysis of divide and conquer algorithms and recursive subroutines
- 3 approaches
  - Iteration Method
  - Master Method
  - Substitution Method

Iteration Method

- Expand the recurrence k times
- Work some algebra to express as a summation
- Evaluate the summation

<u>Examples:</u>

- $T(n) = \begin{cases} T(n-1) + 2, & n > 0 \\ 5, & n = 0 \end{cases}$
- $T(n) = \begin{cases} T(n/2) + 5, & n > 1 \\ 7, & n = 1 \end{cases}$

Master Method (cormen)

In general the Master Theorem says that the recurrence,

$$T(n) = aT(n/b) + f(n)$$

where n/b to mean either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$. T(n) can be bounded asymptotically as follows:

(a) If $f(n) \in O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

(b) If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$

(c) If $f(n) \in \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$

Examples:

(a) $T(n) = 4T(n/2) + n$

(b) $T(n) = 4T(n/2) + n^2$

(c) $T(n) = 4T(n/2) + n^3$

Substitution Method

- Guess the form of the answer
- Use induction to find the constants and show that the solution works.

<u>Example:</u>

Given $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Guess: $T(n) = O(n \log n)$

Prove: $T(n) \leq cn \log n,\ c > 0$

Big-O - Exercise

Given 2 algorithms A1 and A2 performing the same task on n inputs:

A1	A2
$10n$	$n^2/2$
$O(n)$	$O(n^2)$

Which is faster and more efficient?

Big-O - Exercise

Solution:

n	A1	A2
1	10	0.5
5	50	12.5
10	100	50
15	150	112.5
20	200	200
30	300	450

Big-O - Exercise

Arrange the following in increasing order of complexity:

a. n^2, $10n$
b. 4^n, $4n^3$
c. n^2, $n^{1/3}$, n, $n\log_2 n$

Big-O - Exercise

What is the smallest value of *n* such that an algorithm whose running time is *$100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?*

End of Lesson 1

Additional References Used

[1] Katoen, Joost-Pieter. *Introduction to Algorithm Analysis*. http://fmt.cs.utwente.nl/courses/adc/lec1.pdf

[2] *Asymptotic Algorithms Analysis.* http://irl.eecs.umich.edu/jamin/courses/eecs281/winter04/lectures/lecture4j.pdf

[3] Shaof, William. *Asymptotic in Analysis of Algorithms.* http://www.cs.fit.edu/~wds/classes/algorithms/Asym/asymptotics/asymptotics.html