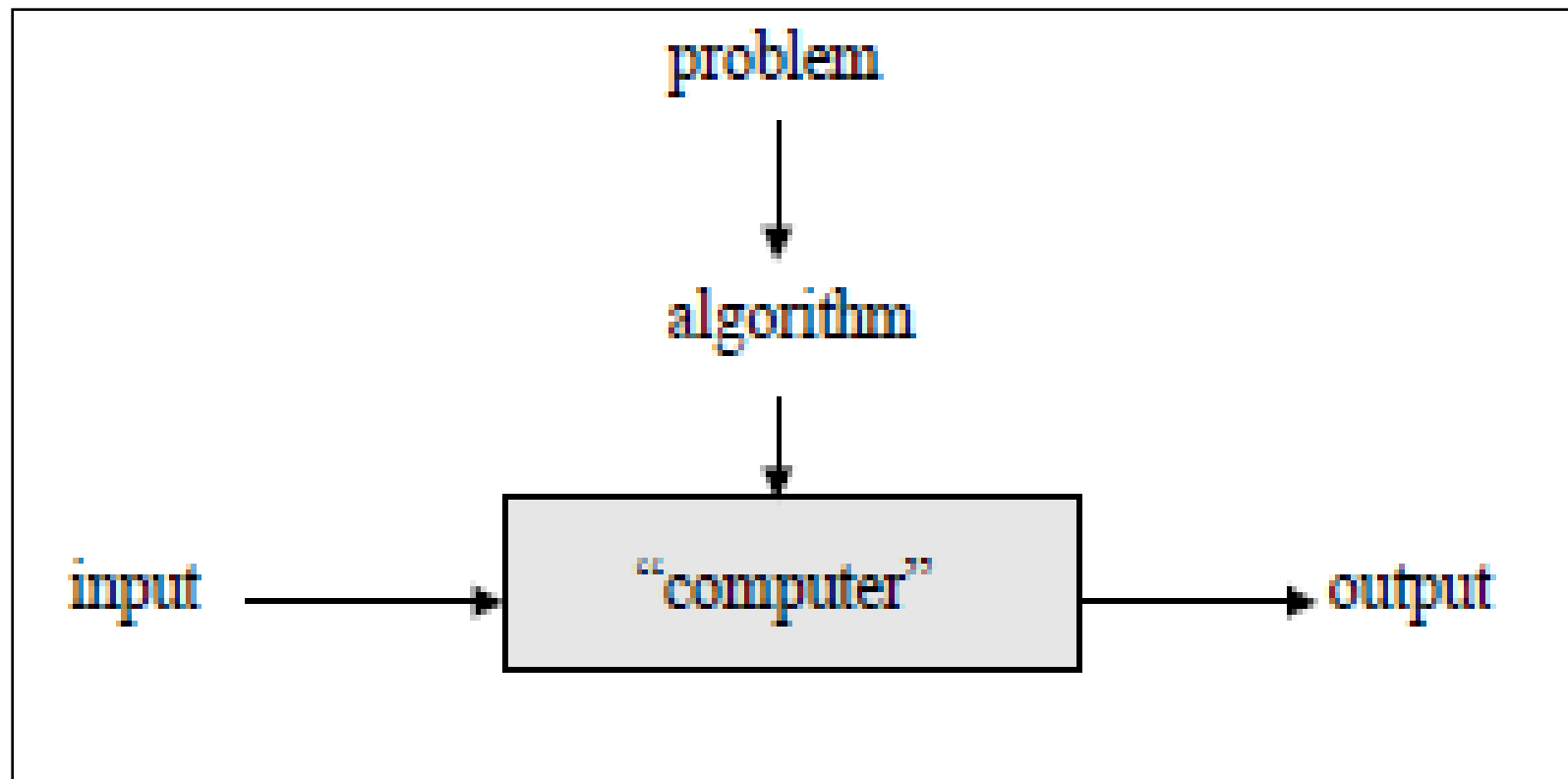


Unit I

Introduction

◆ Algorithm

- ⊕ **An algorithm is any well-defined computational procedure that take 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.**



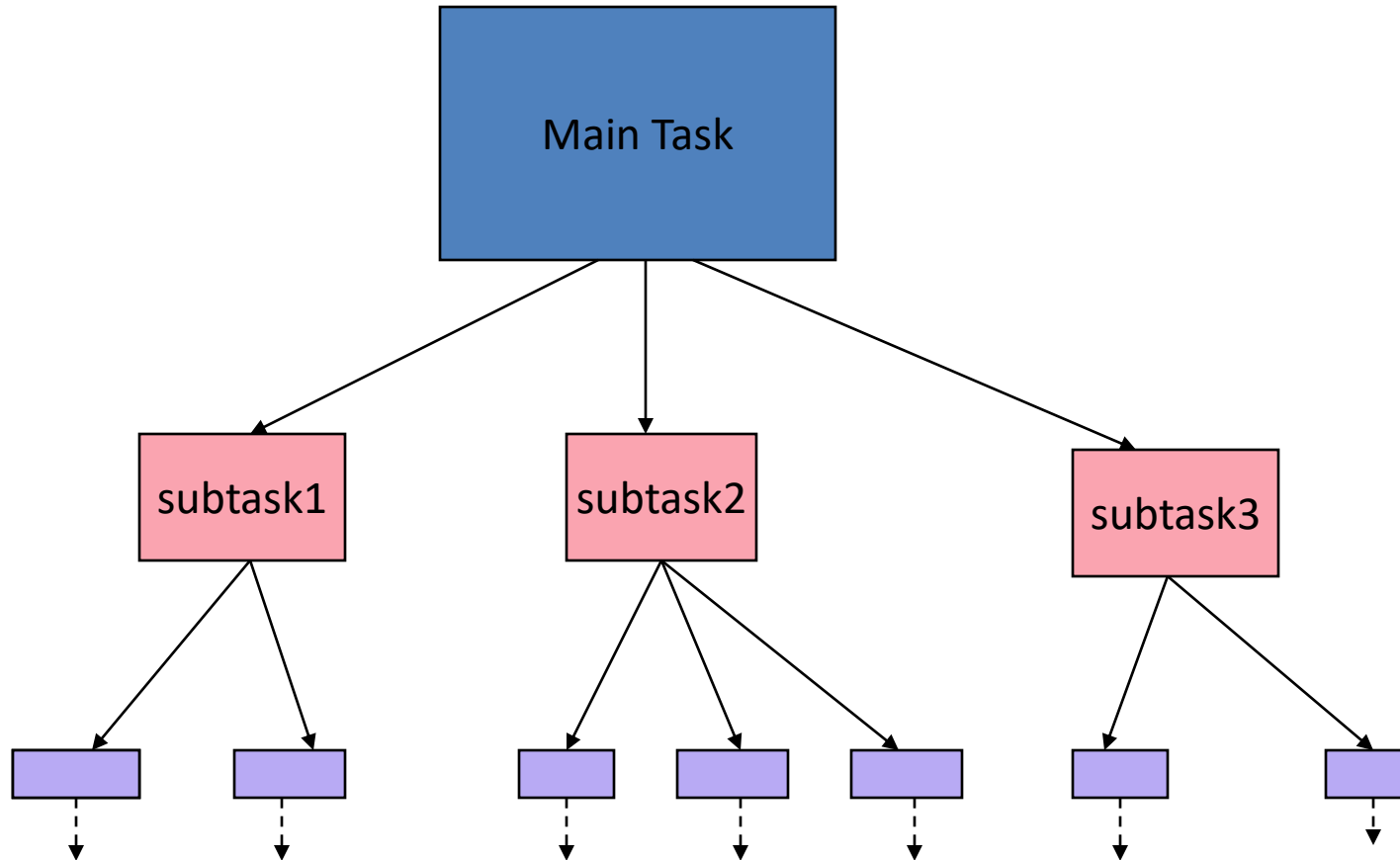
Algorithmic Solution

- With the definition, we can identify five important characteristics of algorithms :
 - Algorithms are well-ordered.
 - Algorithms have unambiguous operations.
 - Algorithms have effectively computable operations.
 - Algorithms produce a result.
 - Algorithms halt in a finite amount of time.

Properties of an Algorithm

- **Finiteness:** - an algorithm terminates after a finite numbers of steps.
- **Definiteness:** - each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion.
- **Input:-** an algorithm accepts zero or more inputs
- **Output:-** it produces at least one output.
- **Effectiveness:-** it consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

Top Down Design



Write an algorithm to find the largest of a set of numbers. You do not know the number of numbers.

FindLargest

Input: A list of positive integers

1. Set Largest to 0
2. while (more integers)
 - 2.1 if (the integer is greater than Largest)
then
 - 2.1.1 Set largest to the value of the integer
 - End if
- End while
3. Return Largest

End

Distinct Areas of Study of Algorithms

◆ Devise an Algorithm

- ⊕ Find good algorithms

◆ Validate a problem

- ⊕ Correct answers for all legal inputs

◆ Analyze an Algorithm

- ⊕ Computational time and memory requirements.

◆ Test a Program

- ⊕ Debugging and profiling

Algorithm Specification

◆ Comments

- ⊕ //

◆ Blocks

- ⊕ { }

◆ Identifier

- ⊕ Starts with letter

◆ Assignment

- ⊕ Id:=4

◆ Boolean Values

- ⊕ True and false

◆ Multidimensional array

- ⊕ A[i,j]. Starts with zero index

◆ Input and Output

- ⊕ Use instructions as read and write

Algorithm Specification

◆ Selection (test)

⊕ *If <condition> then task1*

⊕ *If <condition> then task1 else task2*

⊕ *Multiple cases*

```
case
{
    :<condition 1>: <statement 1>
        ⋮
    :<condition n>: <statement n>
    :else: <statement n + 1>
}
```

Algorithm Specification

Repetition

- While/Repeat

```
While (cond) do  
:  
end while
```

- Do while /Repeat

```
Do  
:  
while (cond)
```

Algorithm Specification

◆ One procedure

Algorithm *Name* (*⟨parameter list⟩*)

⊕ example

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

Example: Selection Sort Algorithm

◆ First Attempt :

- ⊕ Find the smallest from unsorted list and place next to sorted list

```
1  for  $i := 1$  to  $n$  do
2  {
3      Examine  $a[i]$  to  $a[n]$  and suppose
4      the smallest element is at  $a[j]$ ;
5      Interchange  $a[i]$  and  $a[j]$ ;
6  }
```

- ⊕ Smallest element is $a[j]$, then Interchange $a[i]$, $a[j]$

$$t := a[i]; a[i] := a[j]; a[j] := t;$$

Selection Sort

```
1  Algorithm SelectionSort( $a, n$ )
2  // Sort the array  $a[1 : n]$  into nondecreasing order.
3  {
4      for  $i := 1$  to  $n$  do
5      {
6           $j := i$ ;
7          for  $k := i + 1$  to  $n$  do
8              if ( $a[k] < a[j]$ ) then  $j := k$ ;
9               $t := a[i]$ ;  $a[i] := a[j]$ ;  $a[j] := t$ ;
10     }
11 }
```

Recursive Algorithms

◆ Direct Recursion

- ⊕ Function Calling itself

◆ Indirect Recursion

- ⊕ Function A Calling B and B calling A again.

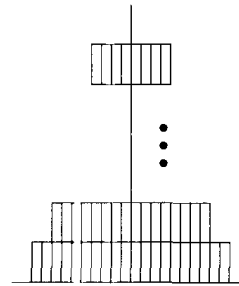
- ⊕ Example: Binomial Theorem

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}$$

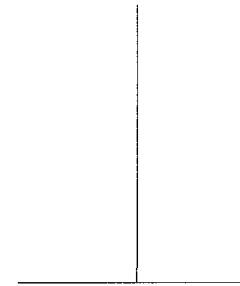
Recursive Algorithms

◆ Examples

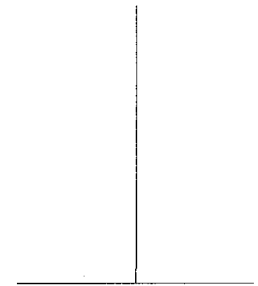
✦ Towers of Hanoi



Tower A



Tower B



Tower C

```
1  Algorithm TowersOfHanoi( $n, x, y, z$ )
2  // Move the top  $n$  disks from tower  $x$  to tower  $y$ .
3  {
4      if ( $n \geq 1$ ) then
5      {
6          TowersOfHanoi( $n - 1, x, z, y$ );
7          write ("move top disk from tower",  $x$ ,
8              "to top of tower",  $y$ );
9          TowersOfHanoi( $n - 1, z, y, x$ );
10     }
11 }
```


Time and Space Complexity

◆ Time complexity

- ⊕ **Time complexity** of an algorithm quantifies the amount of **time** taken by an algorithm to run as a function of the length of the input.

◆ Space complexity

- ◆ **Space complexity** of an algorithm quantifies the amount of **space** or memory taken by an algorithm to run as a function of the length of the input.

Count method to calculate time complexity

- ❖ introduce a new variable, ***count***, ***into a program by*** identifying Program Step and by Increment the value of ***count by appropriate amount with respect*** to a statement in the original program executes.
- ❖ Program Step - A synthetically meaningful segment of a program that requires execution time which is independent on the instance characteristics.

Count method to calculate time complexity...

- ◆ The number of steps in any program depends on the kind of statements. For example –
 - ⊕ – Comments count as zero steps
 - ⊕ – An assignment statement which does not involves any calls to other algorithm is counted as one step.
 - ⊕ – For looping statements the step count equals to the number of step counts assignable to goal value expression. And should be incremented by one within a block and after completion of block.
 - ⊕ – For conditional statements the step count should incremented by one before condition statement.
 - ⊕ – A return statement is counted as one step and should be write before return statement.

◆ **For example –**

⊕ **Algorithm Sum(a,n)**

```
{ s:=0;  
  for i:=1 to n do  
  {  
    s:=s+a[i];  
  }  
  return s;  
}
```

❖ **Algorithm Sum(a,n) // After Adding Count**

```
{   s:=0;
count:=count+1; // for assignment statement execution
for i:=1 to n do
{
    count:=count+1; //for For loop Assignment
    s:=s+a[i];
    count:=count+1;// for addition statement execution
}
count:=count+1; // for last time of for
count:=count+1; // for the return
return s;
}
```

❖ **Algorithm Sum(a,n) //Simplified version for algorithm Sum**

```
{  
    for i:=1 to n do  
    {  
        count:=count+2;  
    }  
    count:=count+3;  
}
```

Form above example,

Total number of program steps= $2n + 3$, where n is the loop counter.

❖ Algorithm RSum(a,n)

{

if $n \leq 0$ then

return $a[n]$;

else

return $\text{RSum}(a, n-1) + a[n]$;

}

◆ Algorithm RSum(a,n)

```
{
    count:=count + 1; // for the if condition
    if  $n \leq 0$  then
    {
        count:=count + 1; // for the return statement
        return a[n];
    }
    else
    {
        count:=count + 1; // for the addition, function invoked
        & return
        return RSum(a, n-1) + a[n];
    }
}
```


◆ Therefore we can write,

- $tRSum(n) = 2$ if $n=0$ and
- $tRSum(n) = 2 + tRSum(n-1)$ if $n>0$
 $= 2 + 2 + tRSum(n-2)$
 $= 2(2) + tRSum(n-2)$
 $= 3(2) + tRSum(n-3)$
 $:$
 $:$
 $= n(2) + tRSum(0)$
 $= 2n+2$

So, the step count for RSum algorithm is $2n+2$.

❖ **Algorithm RSum(a,n)** //Simplified version of algorithm
Rsum with counting's only

```
{  
    count:=count + 1;  
    if  $n \leq 0$  then  
        Count:=count + 1;  
    else  
        count:=count + 1;  
}
```

Table method to calculate time complexity

- ◆ build a ***table in which we list the total number of steps*** contributed by each statement. This table contents three columns –
 - ⊕ ***Steps per execution (s/e)- contents count value by which count*** increases after execution of that statement.
 - ⊕ ***Frequency – is the value indicating total number of times*** statement executes
 - ⊕ ***Total steps – can be obtained by combining s/e and frequency.***
 - ⊕ ***Total step count can be calculated by adding total steps*** contribution values.

Statement	s/e	Frequency	Total steps
Algorithm Sum(a,n)	0	1	0
{	0	1	0
s:=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i];	1	n	n
return s;	1	1	1
}	0	1	0

Total step count = $2n+3$

Analyzing Algorithm

◆ Phases of Analysis

⊕ Priori Analysis

- The bounds of time are obtained by formalating a function based on theroy.
- Independent of programming languages and machine structures. Ex, O-notation.

⊕ Posteriori Analysis

- Depends on programming language and machine structure.
- Time and space is recorded during execution.
- More is the number of input more is the time taken.
Ex. insertion sort

Types of Analysis

❖ Worst case

- ⊕ Provides an upper bound on running time
- ⊕ An absolute **guarantee** that the algorithm would not run longer, no matter what the inputs are

❖ Best case

- ⊕ Provides a lower bound on running time
- ⊕ Input is the one for which the algorithm runs the fastest

$$\textit{Lower Bound} \leq \textit{Running Time} \leq \textit{Upper Bound}$$

❖ Average case

- ⊕ Provides a **prediction** about the running time
- ⊕ Assumes that the input is random

How do we compare algorithms?

- ❖ We need to define a number of objective measures.
 - ⊕ Compare execution times?
 - ***Not good***: times are specific to a particular computer !!
 - ⊕ Count the number of statements executed?
 - ***Not good***: number of statements vary with the programming language as well as the style of the individual programmer.

Ideal Solution

- ❖ Express running time as a function of the input size n (i.e., $f(n)$).
- ❖ Compare different functions corresponding to running times.
- ❖ Such an analysis is independent of machine time, programming style, etc.

Example

- ◆ Associate a "cost" with each statement.
- ◆ Find the "total cost" by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	...
arr[N-1] = 0;	c_1

$$\text{-----}$$
$$c_1 + c_1 + \dots + c_1 = c_1 \times N$$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$$\text{-----}$$
$$(N+1) \times c_2 + N \times c_1 =$$
$$(c_2 + c_1) \times N + c_2$$

Another Example

❖ *Algorithm 3*

Cost

sum = 0;

c_1

for(i=0; i<N; i++)

c_2

for(j=0; j<N; j++)

c_2

sum += arr[i][j];

c_3

$$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$$

Asymptotic Notation

◆ **O notation: asymptotic “less than”:**

⊕ $f(n) = O(g(n))$ implies: $f(n) \leq g(n)$

◆ **Ω notation: asymptotic “greater than”:**

⊕ $f(n) = \Omega(g(n))$ implies: $f(n) \geq g(n)$

◆ **Θ notation: asymptotic “equality”:**

⊕ $f(n) = \Theta(g(n))$ implies: $f(n) = g(n)$

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.

More Examples ...

◆ $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$

◆ $10n^3 + 2n^2$ is $O(n^3)$

◆ $n^3 - n^2$ is $O(n^3)$

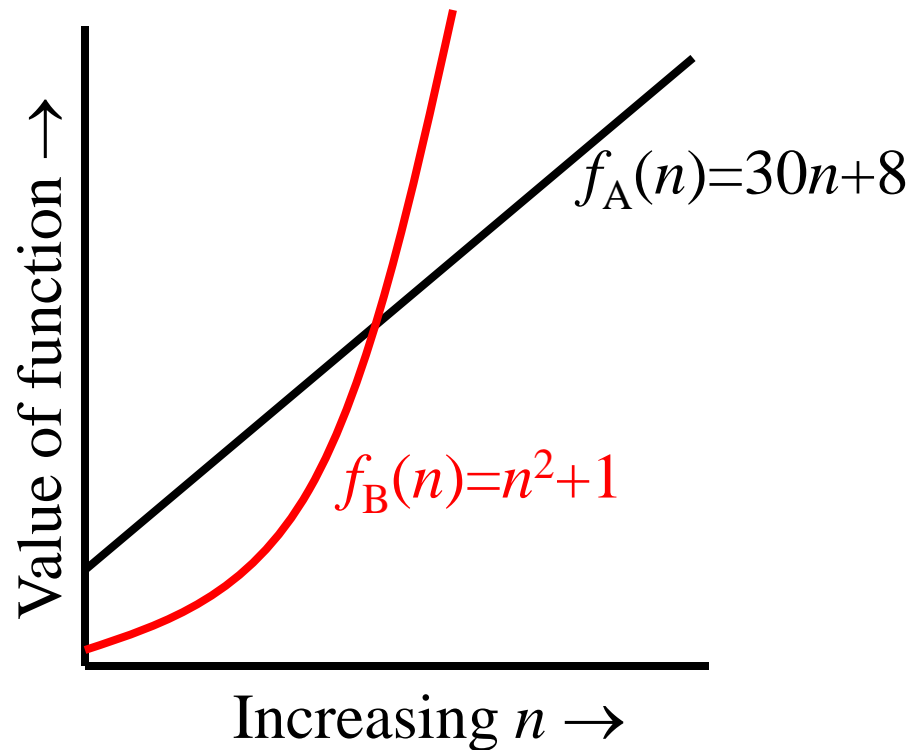
◆ constants

⊕ 10 is $O(1)$

⊕ 1273 is $O(1)$

Visualizing Orders of Growth

- ◆ On a graph, as you go to the right, a faster growing function eventually becomes larger...



Back to Our Example

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	
arr[N-1] = 0;	c_1

$$c_1 + c_1 + \dots + c_1 = c_1 \times N$$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$$(N+1) \times c_2 + N \times c_1 =$$
$$(c_2 + c_1) \times N + c_2$$

◆ Both algorithms are of the same order: $O(N)$

Example (cont'd)

Algorithm 3

sum = 0;

for(i=0; i<N; i++)

 for(j=0; j<N; j++)

 sum += arr[i][j];

Cost

c_1

c_2

c_2

c_3

$$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2 = O(N^2)$$

Review: Asymptotic Performance

❖ *Asymptotic performance*: How does algorithm behave as the problem size gets very large?

- Running time
- Memory/storage requirements

⊕ Remember that we use the RAM model:

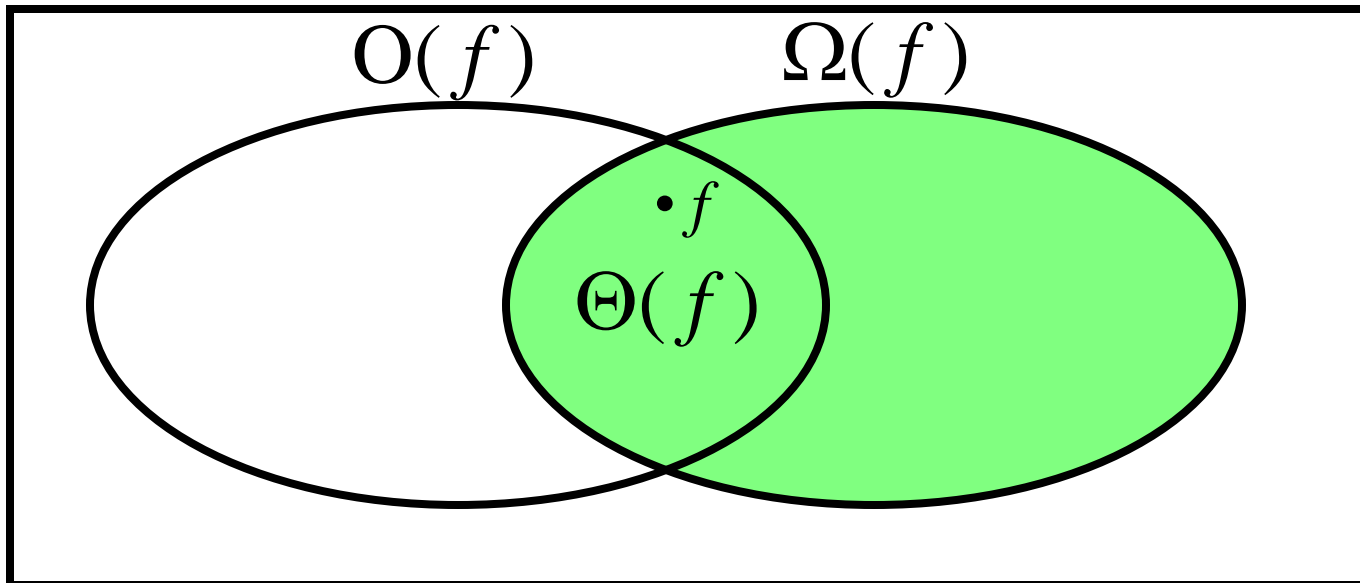
- All memory equally expensive to access
- No concurrent operations
- All reasonable instructions take unit time
 - Except, of course, function calls
- Constant word size
 - Unless we are explicitly manipulating bits

Review: Running Time

◆ Number of primitive steps that are executed

- ◆ Except for time of executing a function call most statements roughly require the same amount of time We can be more exact if need be

$\mathbf{R} \rightarrow \mathbf{R}$



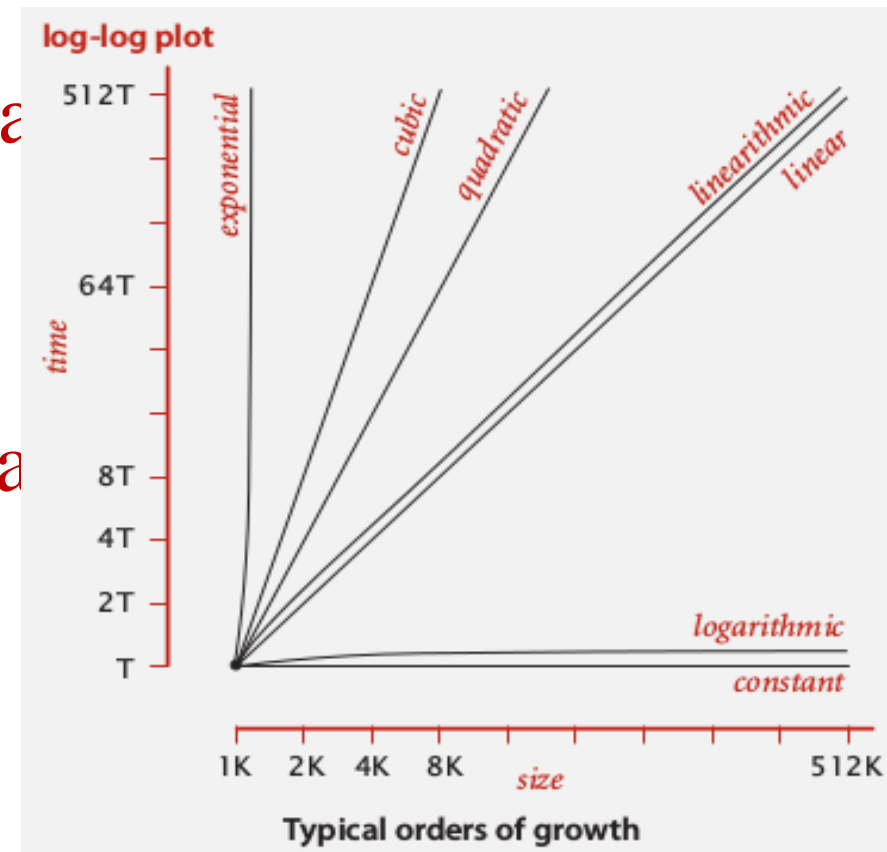
Asymptotic Notations

◆ Allow us

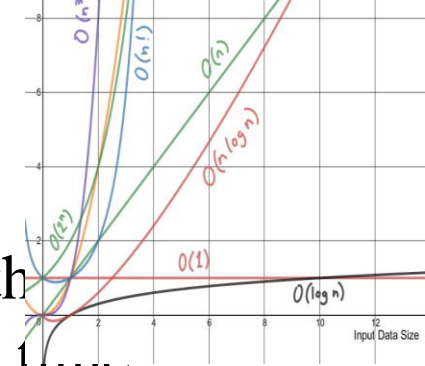
◆ to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.

◆ This is also known as an Θ rate.

◆ Order of Growth classification



Asymptotic Notations



Constant

- ⊕ No matter the size of the data it receives, the algorithm takes the same amount of time to run. We denote this as a time complexity of $O(1)$.

Linear

- ⊕ The running duration of a linear algorithm is constant. It will process the input in n number of operations $O(n)$.

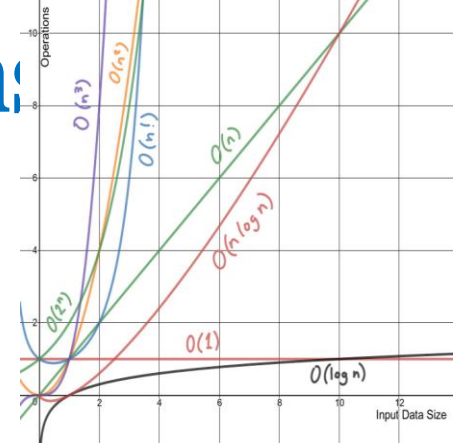
Quadratic

- ⊕ two nested loops, or nested linear operations, the algorithm processes the input n^2 times.

Logarithmic

- ⊕ A logarithmic algorithm is one that reduces the size of the input at every step. We denote this time complexity as $O(\log n)$. Example : binary search algorithm

Asymptotic Notation:



- **Quasilinear**
 - the time complexity $O(n \log n)$ can describe a data structure where each operation takes $O(\log n)$ time. example :quick sort, a divide-and-conquer algorithm.
- **Non-polynomial time complexity**
 - An algorithm with time complexity $O(n!)$ often iterates through all permutations of the input elements. example brute-force search seen in the travelling salesman problem
- **Exponential**
 - An exponential algorithm often also iterates through all subsets of the input elements. It is denoted $O(2^n)$. The larger the data set, the more steep the curve becomes. a brute-force attack.

Order of Growth classification

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	<pre>a = b + c;</pre>	statement	add two numbers	1
$\log N$	logarithmic	<pre>while (N > 1) { N = N / 2; ... }</pre>	divide in half	binary search	~ 1
N	linear	<pre>for (int i = 0; i < N; i++) { ... }</pre>	loop	find the maximum	2
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }</pre>	double loop	check all pairs	4
N^3	cubic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }</pre>	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

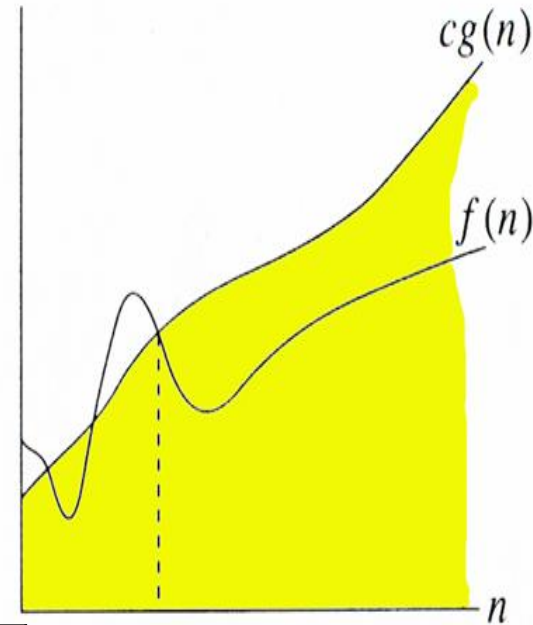
Asymptotic Notations- Big O Notation

$$\{f(n) = O(g(n))\}$$

$f(n) \leq cg(n)$ for all $n \geq n_0$

there exist positive constants $c > 0$ and $n_0 \geq 1$

Example:



$$n_0 \quad f(n) = O(g(n))$$

$f(n)=3n+5$	$f(n)=27n^2+16n$	$f(n)=2n^3+6n^2+2n$
$3n+5 \leq$ $3n+n \leq 4n$ $n \geq 5$	$27n^2+16n \leq 27n^2+n^2$ $\{n \leq n^2\}$ $27n^2+16n \leq 28n^2$	
$f(n) = O(n)$	$f(n) = O(n^2)$	

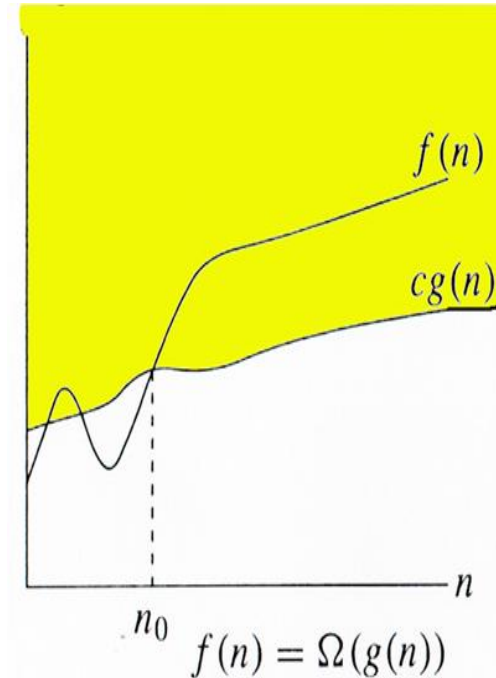
Asymptotic Notations- Big Omega (Ω) Notation

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

$cg(n) \leq f(n)$ for all $n \geq n_0$
 there exist positive constants $c > 0$ and $n_0 \geq 1$

Example:

$f(n) = 3n + 5$	$f(n) = 27n^2 + 16n$	$f(n) = 2n^3 + 6n^2 + 2n$
$3n \leq 3n + 5$	$27n^2 \leq 27n^2 + 16n$	
$f(n) = \Omega(n)$	$f(n) = \Omega(n^2)$	

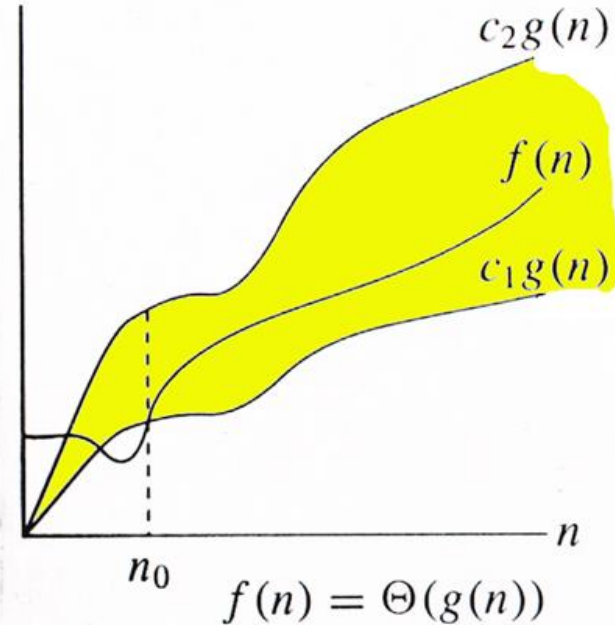


Asymptotic Notations- Big Theta (Θ) Notation

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

**$c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$
there exist positive constants $c_1, c_2 > 0$ and
 $n_0 \geq 1$**

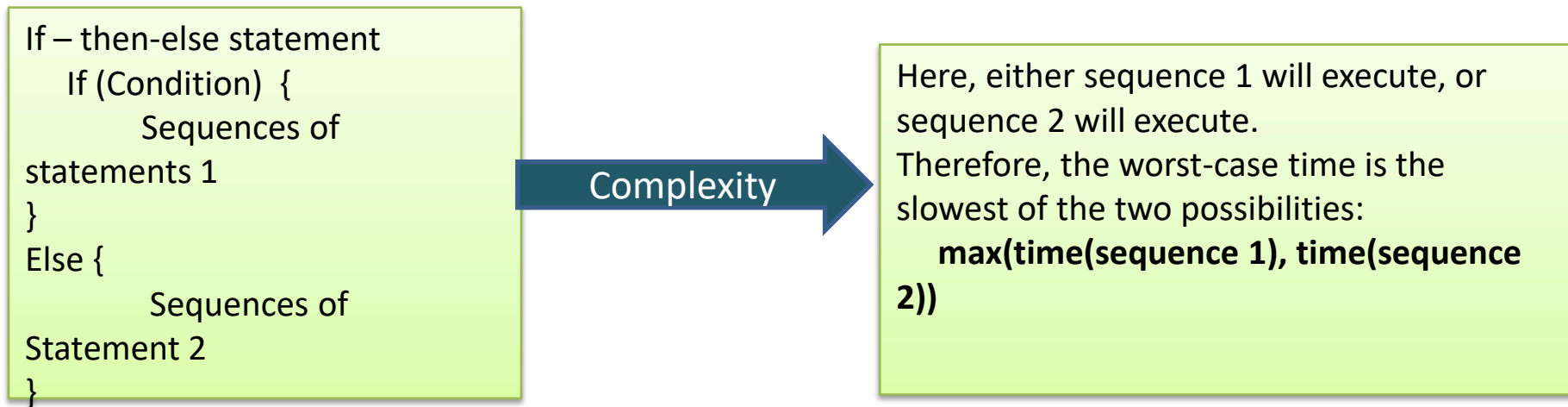
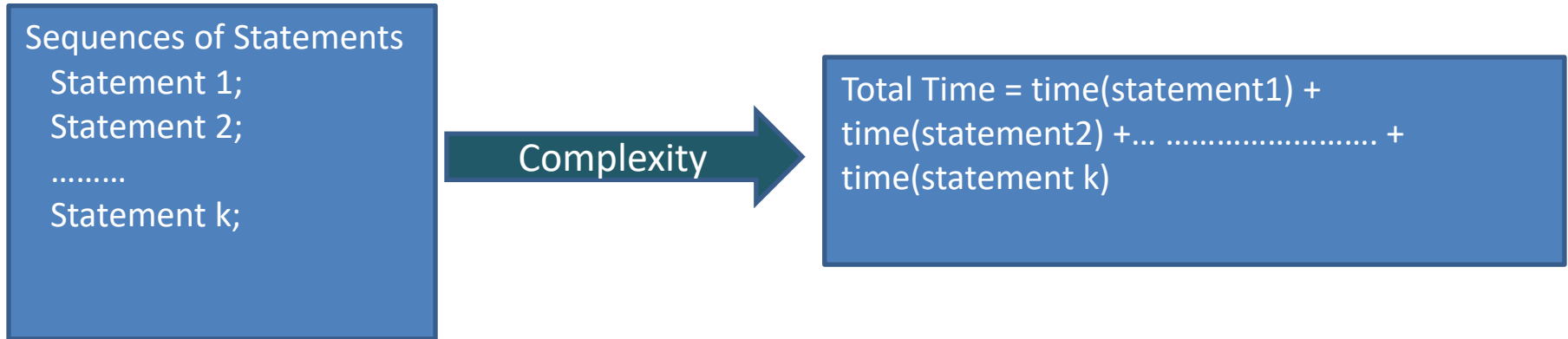
Example:



$f(n)=3n+5$	$f(n)=27n^2+16n$	$f(n)=2n^3+6n^2+2n$
$3n \leq 3n+n \leq 4n$ $C1=3$ & $c2=4$	$27n^2+16n \leq 27n^2+n$ $\{n \leq n^2\}$ $27n^2 \leq 27n^2+16n \leq 28n^2$	
$f(n) = \Theta(n)$	$f(n) = \Theta(n^2)$	

How to Determine Complexities

- ❖ In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.



How to Determine Complexities

```
for loops
for (i = 0; i < N; i++) {
    {
        sequence of
        statements
    }
}
```

Complexity

The loop executes N times, so the sequence of statements also executes N times. which is **$O(N)$** overall.

Nested loops

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
    }
}
```

Complexity

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times
Thus, the complexity is **$O(N * M)$**

```
x = 0;
A[n] = some array of
length n;
while (x != A[i])
{
    i++;
}
```

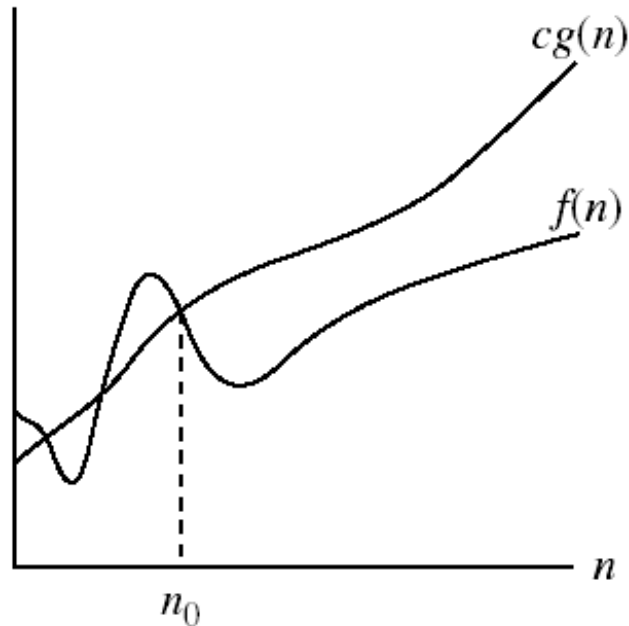
Complexity

The loop executes N times, so the sequence of statements also executes N times. which is **$O(N)$** overall.

Asymptotic notations

◆ *O-notation*

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



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

Examples — ('=' symbol readed as 'is' instead of 'equal')

- ◆ The function $3n+2 = O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$
- ◆ The function $3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$
- ◆ The function $100n+6 = O(n)$ as $100n+6 \leq 101n$ for all $n \geq 6$
- ◆ The function $10n^2+4n+2 = O(n^2)$ as $10n^2+4n+2 \leq 11n^2$ for all $n \geq 5$
- ◆ The function $1000n^2+100n-6 = O(n^2)$ as $1000n^2+100n-6 \leq 1001n^2$ for all $n \geq 100$
- ◆ The function $6 \cdot 2^n + n^2 = O(2^n)$ as $6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$ for all $n \geq 4$
- ◆ The function $3n+3 = O(n^2)$ as $3n+3 \leq 3n^2$ for all $n \geq 2$

Tabular Method

n	Function f(n)	compare	c. g(n)
	$10n^2+4n+2$		$11n^2$
1	$10+4+2=16$	>	11
2	$40+8+2=50$	>	44
3	$90+12+2=104$	>	99
4	$160+16+2=178$	>	176
5	$250+20+2=272$	<	275
6	$360+24+2=386$	<	396

❖ Consider the job offers from two companies. The first company offer contract that will double the salary every year. The second company offers you a contract that gives a raise of Rs. 1000 per year. This scenario can be represented with Big-O notation as –

- ⊕ For first company, New salary = $\text{Salary} \times 2^n$ (where n is total service years)
- ⊕ Which can be denoted with Big-O notation as $O(2^n)$
- ⊕ For second company, New salary = $\text{Salary} + 1000n$ (where n is total service years)
- ⊕ Which can be denoted with Big-O notation as $O(n)$

O(1)

- ❖ Describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set i.e. a computing time is a constant time.

```
int IsFirstElementNull(char String[])
{
    if(strings[0] == '\0')
    {
        return 1;
    }
    return 0;
}
```


$O(N)$: is called *linear time*,

- Describe an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

```
int ContainsValue(char String[], int no, char ch)
{
    for( i = 0; i < no; i++)
    {
        if(string[i] == ch)
        {
            return 1;
        }
    }
    return 0;
}
```

$O(N^k)$: (k fixed) refers to *polynomial time*; (if $k=2$, it is called *quadratic time*, $k=3$, it is called *cubic time*),

◆ which represents an algorithm whose performance is directly proportional to the square of the size of the input data set.

⊕ This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.

```
bool ContainsDuplicates(String[] strings)
{
    for(int i = 0; i < strings.Length; i++)
    {
        for(int j = 0; j < strings.Length; j++)
        {
            if(i == j) // Don't compare with self
                continue;
            if(strings[i] == strings[j])
                return true;
        }
        return false;
    }
}
```

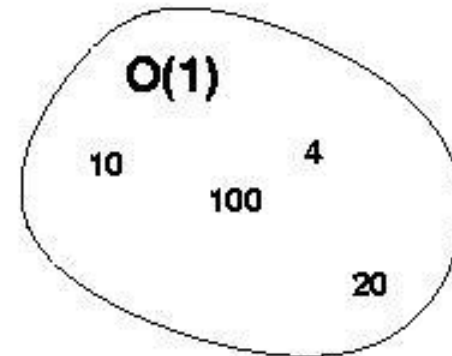
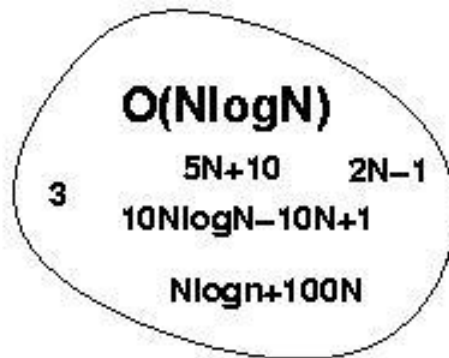
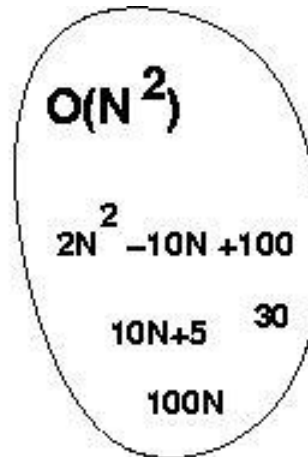
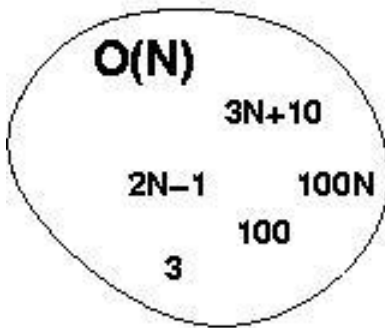
$O(2^N)$: is called *exponential time*,

❖ Denotes an algorithm whose growth will double with each additional element in the input data set.

⊕ The execution time of an $O(2^N)$ function will quickly become very large.

Big-O Visualization

$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$



An Example: Insertion Sort

```
InsertionSort(A, n) {  
    for i = 2 to n {  
        key = A[i]  
        j = i - 1;  
        while (j > 0) and (A[j] > key) {  
            A[j+1] = A[j]  
            j = j - 1  
        }  
        A[j+1] = key  
    }  
}
```

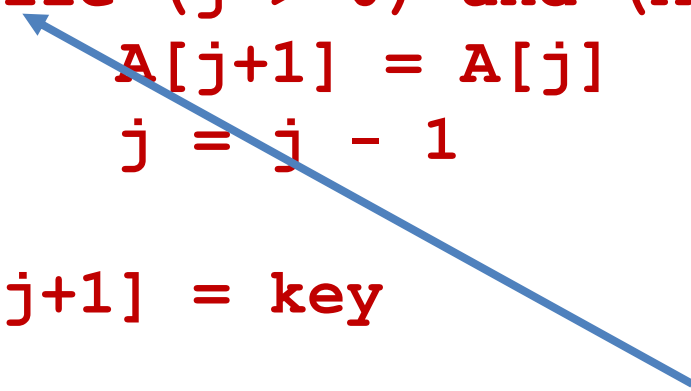
Insertion Sort

```
InsertionSort(A, n) {  
    for i = 2 to n {  
        key = A[i]  
        j = i - 1;  
        while (j > 0) and (A[j] > key) {  
            A[j+1] = A[j]  
            j = j - 1  
        }  
        A[j+1] = key  
    }  
}
```

What is the precondition for this loop?

Insertion Sort

```
InsertionSort(A, n) {  
    for i = 2 to n {  
        key = A[i]  
        j = i - 1;  
        while (j > 0) and (A[j] > key) {  
            A[j+1] = A[j]  
            j = j - 1  
        }  
        A[j+1] = key  
    }  
}
```



How many times will this loop execute?

Insertion Sort

Statement	Effort
InsertionSort (A, n) {	
for i = 2 to n {	$c_1 n$
key = A[i]	$c_2(n-1)$
j = i - 1;	$c_3(n-1)$
while (j > 0) and (A[j] > key) {	$c_4 T$
A[j+1] = A[j]	$c_5(T-(n-1))$
j = j - 1	$c_6(T-(n-1))$
}	0
A[j+1] = key	$c_7(n-1)$
}	0
}	

$T = t_2 + t_3 + \dots + t_n$ where t_i is number of while expression evaluations for the i^{th} for loop iteration

Analyzing Insertion Sort

◆ $T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$
 $= c_8T + c_9n + c_{10}$

◆ What can T be?

⊕ Best case -- inner loop body never executed

- $t_i = 1 \rightarrow T(n)$ is a linear function

⊕ Worst case -- inner loop body executed for all previous elements

- $t_i = i \rightarrow T(n)$ is a quadratic function

⊕ Average case

- ???

Upper Bound Notation

- ◆ We say InsertionSort's run time is $O(n^2)$
 - ⊕ Properly we should say run time is *in* $O(n^2)$
 - ⊕ Read O as “Big- O ” (you'll also hear it as “order”)
- ◆ In general a function
 - ⊕ $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- ◆ Formally
 - ⊕ $O(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \forall n \geq n_0 \}$

Insertion Sort Is $O(n^2)$

◆ Proof

- ⊕ Suppose runtime is $an^2 + bn + c$
 - If any of a , b , and c are less than 0 replace the constant with its absolute value
- ⊕ $an^2 + bn + c \leq (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
- ⊕ $\leq 3(a + b + c)n^2$ for $n \geq 1$
- ⊕ Let $c' = 3(a + b + c)$ and let $n_0 = 1$

◆ Question

- ⊕ Is InsertionSort $O(n^3)$?
- ⊕ Is InsertionSort $O(n)$?

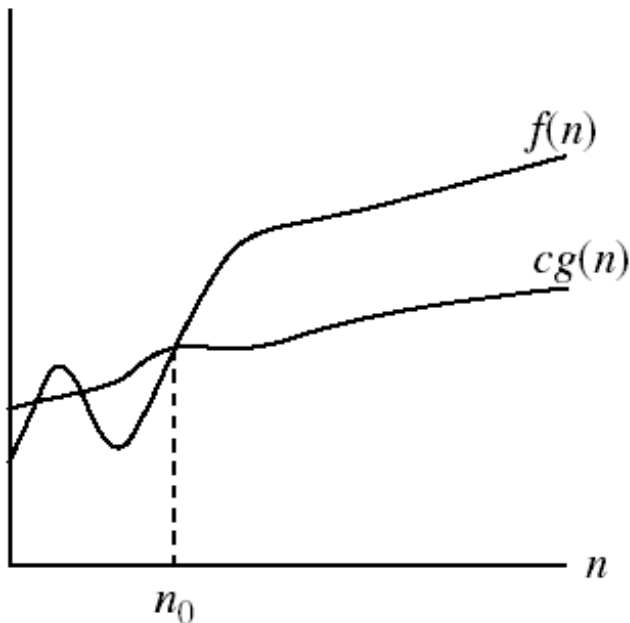
Omega notation (Ω)

- ❖ specifically describes the **best-case scenario**, and can be used to describe the minimum execution time (asymptotic lower bounds) required or the space used (e.g. in memory or on disk) by an algorithm.
- ❖ The function $f(n) = \Omega(g(n))$ (read as '*f of n is said to be Omega of g of n*') if and only if there exists a real, positive constant C and a positive integer n_0 such that, $f(n) \geq C * g(n)$ for all $n \geq n_0$
- ❖ Here, n_0 must be greater than 0.

Asymptotic notations (cont.)

◆ Ω - notation

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



$\Omega(g(n))$ 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 –

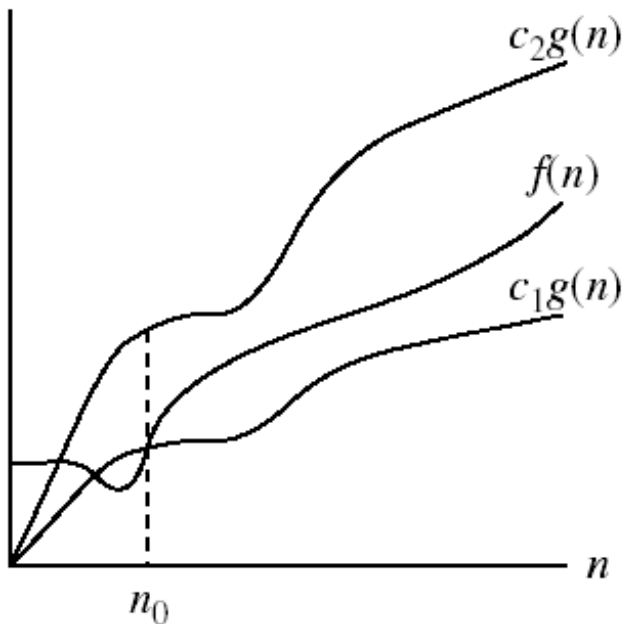
- The function $3n+2 = \Omega(n)$ as $3n+2 \geq 3n$ for all $n \geq 1$
- The function $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for all $n \geq 1$
- The function $100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for all $n \geq 1$
- The function $10n^2+4n+2 = \Omega(n^2)$ as $10n^2+4n+2 \geq n^2$ for all $n \geq 1$
- The function $6 \cdot 2^n + n^2 = \Omega(2^n)$ as $6 \cdot 2^n + n^2 \geq 2^n$ for all $n \geq 1$
- The function $3n+3 = \Omega(1)$
- The function $10n^2+4n+2 = \Omega(n)$
- The function $10n^2+4n+2 = \Omega(1)$

Theta notation (Θ)

- ❖ Theta specifically describes the **average-case** scenario, and can be used to describe the average execution time required or the space used by an algorithm.
- ❖ A description of a function in terms of Θ notation usually only provides an tight bound on t
- ❖ The function $f(n) = \Theta(g(n))$ (read as 'f of n is said to be Theta of g of n') if and only if there exists a positive constants $C1$, $C2$, and a positive integer n_0 such that, $C1 * g(n) \leq f(n) \leq C2 * g(n)$ for all $n \geq n_0$

Asymptotic notations (cont.)

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\} .$



$\Theta(g(n))$ 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

- The function $3n+2 = \Theta(n)$ as
 $3n+2 \geq 3n$ for all $n \geq 2$ and
 $3n+2 \leq 4n$ for all $n \geq 2$

So, $c_1=3$, $c_2=4$ and $n_0=2$.

- The function $3n+3 = \Theta(n)$
- The function $10n^2+4n+2 = \Theta(n^2)$
- The function $6 * 2^n + n^2 = \Theta(2^n)$

Next Lecture

◆ Recurrence Relations

Recurrence Relations

- ❖ **Definition:** Given a recursive algorithm a recurrence relation for the algorithm is an equation that gives the run time on an input size in terms of the run times of smaller input sizes.
- ❖ When iterative formulas for $T(n)$ are difficult or impossible to obtain, one can use either
 - ⊕ a recursion tree method,
 - ⊕ an iteration method , or
 - ⊕ a substitution method with Induction to get $T(n)$ or a bound $U(n)$ of $T(n)$, where $T(n) = \Theta(U(n))$.

Recursion Tree

- ◆ A recursion tree is a tree generated by tracing the execution of a recursive algorithm.

Recurrence Relations

◆ *A recurrence relation for the sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more of the previous terms of the sequence, namely, a_0, a_1, \dots, a_n for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer. A sequence is called a solution of a recurrence relation if its terms satisfy the recurrence relation.*

◆ Let $\{a_n\}$ be a sequence that satisfies the recurrence relation $a_n = a_{n-1} + a_{n-2}$ for $n = 1, 2, 3, \dots$, and suppose that $a_0 = 2$. What are a_1 , a_2 , and a_3 ?

Solution: We see from the recurrence relation that

$$a_1 = a_0 + 3 = 2 + 3 = 5.$$

$$a_2 = 5 + 3 = 8 \quad \text{and}$$

$$a_3 = 8 + 3 = 11$$

◆ Let $\{a_n\}$ be a sequence that satisfies the recurrence relation $a_n = a_{n-1} - a_{n-2}$ for $n = 2, 3, 4, \dots$, and suppose that $a_0 = 3$ and $a_1 = 5$. What are a_2 , and a_3 ?

Solution: We see from the recurrence relation that $a_2 = a_1 - a_0 = 5 - 3 = 2$

and $a_3 = a_2 - a_1 = 2 - 5 = -3$

We can find a_4 , a_5 , and each successive term in a similar way.

Creating Recurrence Relation

Fib(a)	$T(n)$
{	
if(a==1 a==0)	1
return 1;	
return Fib(a-1) + Fib(a-2);	$T(n-1)+T(n-2)$
}	

(comparison, comparison, addition) and also calls itself recursively.

$$f_n = f_{n-1} + f_{n-2}$$

The *Fibonacci sequence*, f_0, f_1, f_2, \dots , is defined by the initial conditions $f_0 = 0$, $f_1 = 1$, and the recurrence relation $f_n = f_{n-1} + f_{n-2}$ for $n = 2, 3, 4, \dots$.

◆ Find the Fibonacci numbers f_2, f_3, f_4, f_5 , and f_6 .

Solution:. Because the initial conditions tell us that $f_0 = 0$ and $f_1 = 1$, using the recurrence relation in the definition we find that

◆ $f_2 = f_1 + f_0 = 1 + 0 = 1,$

◆ $f_3 = f_2 + f_1 = 1 + 1 = 2,$

◆ $f_4 = f_3 + f_2 = 2 + 1 = 3,$

◆ $f_5 = f_4 + f_3 = 3 + 2 = 5,$

◆ $f_6 = f_5 + f_4 = 5 + 3 = 8.$

Solving Recurrence Relations

A recurrence relation is an equation that recursively defines a sequence where the next term is a function of the previous terms (Expressing F_n as some combination of F_i with $i < n$).

Recurrence Relations

◆ Substitution Method

⊕ The *substitution method for solving recurrences comprises two steps:*

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

⊕ We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name “substitution method.”

⊕ This method is powerful, but we must be able to guess the form of the answer in order to apply it.

Example

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

- ◆ We guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding

$$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor).$$

- ◆ Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n , \end{aligned}$$

where the last step holds as long as $c \geq 1$.

$$T(n) = \begin{cases} 3T(n-1), & \text{if } n > 0, \\ 1, & \text{otherwise} \end{cases}$$

Let us solve using substitution.

$$\begin{aligned} T(n) &= 3T(n-1) = 3(3T(n-2)) \\ &= 3^2T(n-2) \\ &= 3^3T(n-3) \dots \dots \\ &= 3^nT(n-n) \\ &= 3^nT(0) \\ &= 3^n \end{aligned}$$

This clearly shows that the complexity of this function is $O(3^n)$.

How to solve linear recurrence relation

Homogeneous Recurrence Relations

- ◆ Suppose, a two ordered linear recurrence relation is $F_n = AF_{n-1} + BF_{n-2}$
 ◆ where A and B are real numbers.
- ◆ The characteristic equation for the above recurrence relation is –

$$x^2 - Ax - B = 0$$
- ◆ Three cases may occur while finding the roots –
- ◆ **Case 1** – If this equation factors as $(x - x_1)(x - x_1) = 0$
 and it produces two distinct real roots x_1 and x_2 , then $F_n = ax_1^n + bx_2^n$
 is the solution. [Here, a and b are constants]
- ◆ **Case 2** – If this equation factors as $(x - x_1)^2$
 and it produces single real root x_1 , then $F_n = ax_1^n + bnx_1^n$
 is the solution.
- ◆ **Case 3** – If the equation produces two distinct complex roots, x_1
 and x_2 in polar form $x_1 = r\angle\Theta$ and $x_2 = r\angle(-\Theta)$, then
 following is the solution

$$F_n = r^n (a \cos(n\theta) + b \sin(n\theta))$$

Problem 1

Solve the recurrence relation $F_n = 5F_{n-1} - 6F_{n-2}$
where $F_0 = 1$ and $F_1 = 4$

Solution

◆ The characteristic equation of the recurrence relation is – $x^2 - 5x + 6 = 0$

So, $(x-3)(x-2) = 0$

Hence, the roots are –

$$x_1 = 3$$

and $x_2 = 2$

The roots are real and distinct. So, this is in the form of case 1

◆ Hence, the solution is –

$$F_n = ax_1^n + bx_2^n$$

◆ Here, $F = a3^n + b2^n$ (As $x_1=3$ and $x_2=2$)

◆ Therefore,

$$1 = F_0 = a3^0 + b2^0 = a + b$$

$$4 = F_1 = a3^1 + b2^1 = 3a + 2b$$

◆ Solving these two equations, we get $a=2$ and $b=-1$

◆ Hence, the final solution is –

$$F_n = 2.3^n + (-1).2^n = 2.3^n - 2^n$$

Problem 2

Solve the recurrence relation $F_n = 10F_{n-1} - 25F_{n-2}$
where $F_0 = 3$ and $F_1 = 17$

◆ Solution

◆ The characteristic equation of the recurrence relation is –
$$x^2 - 10x - 25 = 0$$

◆ So $(x - 5)^2 = 0$

◆ Hence, there is single real root $x_1 = 5$

◆ As there is single real valued root, this is in the form of case 2

◆ Hence, the solution is –

$$F_n = ax_1^n + bnx_1^n$$

◆ $3 = F_0 = a.5^0 + b.0.5^0$ & $17 = F_1 = a.5^1 + b.1.5^1$

◆ Solving these two equations, we get $a=3$

◆ and $b=2/5$

◆ Hence, the final solution is
$$F_n = 3.5^n + (2/5).n.2^n$$

Problem 3

Solve the recurrence relation $F_n = 2F_{n-1} - 2F_{n-2}$
where $F_0 = 1$ and $F_1 = 3$

◆ Solution

◆ The characteristic equation is $x^2 - 2x - 2x = 0$
Hence, the roots are – $x_1 = 1 + i$ and $x_2 = 1 - i$

◆ In polar form, $x_1 = r \angle \theta$ And $x_2 = r \angle (-\theta)$
where $r = \sqrt{2}$ and $\theta = \frac{\pi}{4}$

◆ The roots are imaginary. So, this is in the form of case 3.

◆ Hence,

$$F_n = (\sqrt{2})^n (a \cos(n \cdot \frac{\pi}{4}) + b \sin(n \cdot \frac{\pi}{4}))$$
$$1 = F_0 = (\sqrt{2})^0 (a \cos(0 \cdot \frac{\pi}{4}) + b \sin(0 \cdot \frac{\pi}{4})) = a$$
$$3 = F_1 = (\sqrt{2})^1 (a \cos(1 \cdot \frac{\pi}{4}) + b \sin(1 \cdot \frac{\pi}{4})) = \sqrt{2}(a / \sqrt{2} + b \sqrt{2})$$

Solving these two equations we get $a=1$ and $b=2$

◆ Hence, the final solution is –

$$F_n = (\sqrt{2})^n (\cos(n \cdot \frac{\pi}{4}) + 2 \sin(n \cdot \frac{\pi}{4}))$$

Example: Fibonacci sequence

$$f_n = f_{n-1} + f_{n-2}$$

$$f_0 = 0, \quad f_1 = 1$$

Has solution:
$$f_n = \lambda_1 r_1^n + \lambda_2 r_2^n$$

Characteristic roots:
$$r_1 = \frac{1 + \sqrt{5}}{2} \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

$$\lambda_1 = \frac{f_1 - f_0 r_2}{r_1 - r_2} = \frac{1}{\sqrt{5}}$$

$$\lambda_2 = \frac{f_0 r_1 - f_1}{r_1 - r_2} = -\frac{1}{\sqrt{5}}$$

$$f_n = \lambda_1 r_1^n + \lambda_2 r_2^n$$

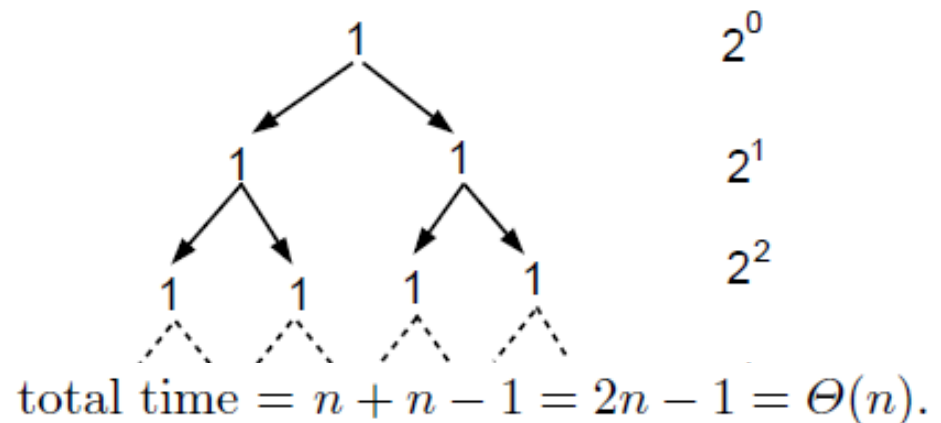
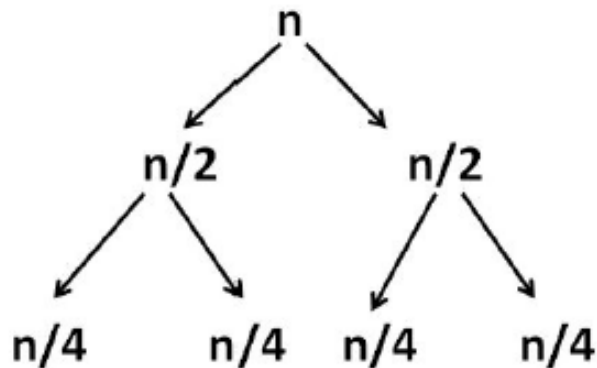
$$= \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

Recursion Tree (Back Substitution)

$$T(n) = 2T(n/2) + 1$$

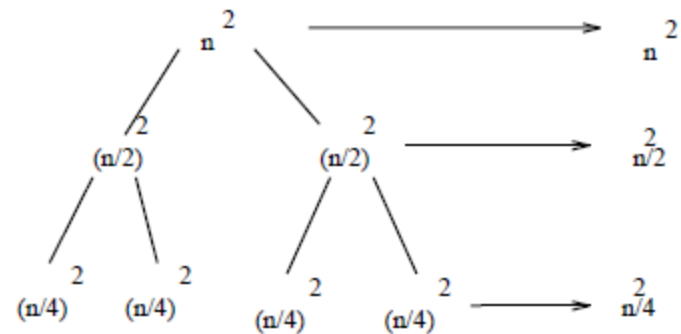
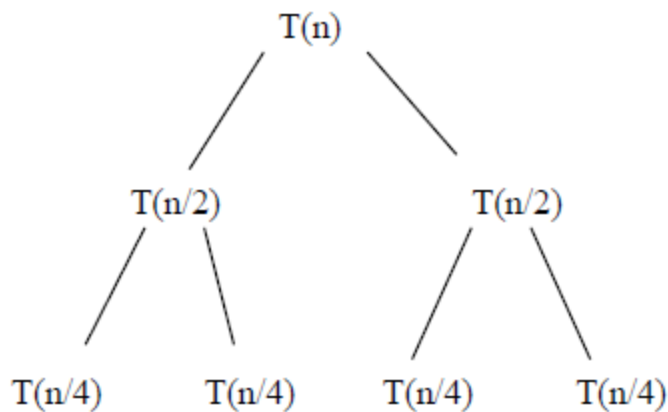
Here the number of leaves = $2^{\log n} = n$

and the sum of effort in each level except leaves is $\sum_{i=0}^{\log_2(n)-1} 2^i = 2^{\log_2(n)} - 1 = n - 1$



Recursion Tree (Back Substitution)

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



$$T(n) = \sum_{i=0}^{\infty} n^2/2^i = n^2 \sum_{i=0}^{\infty} 1/2^i = \Theta(n^2)$$

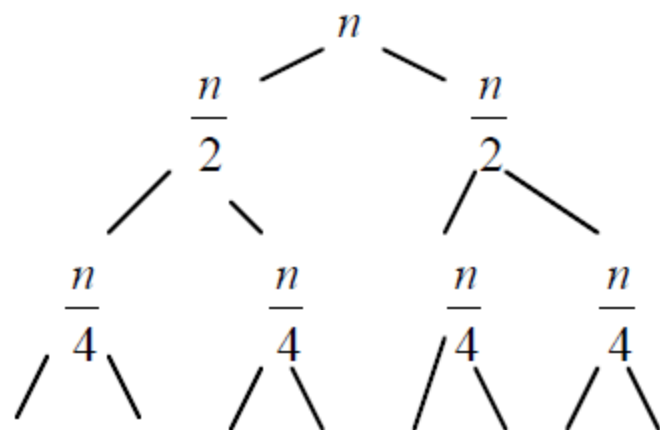
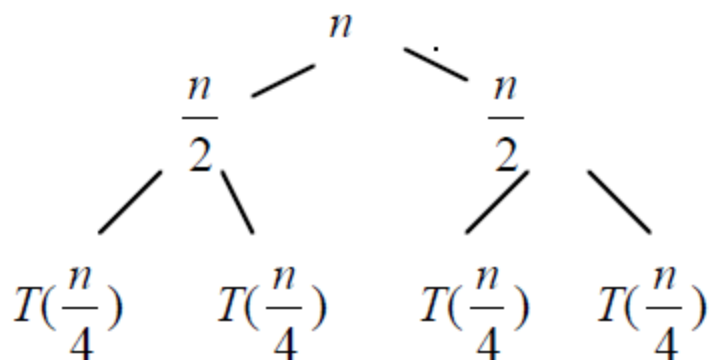
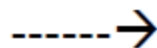
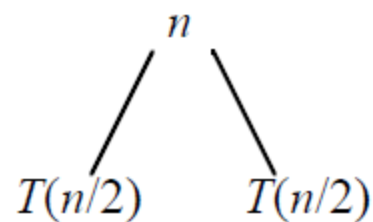
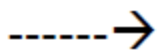


$$T(n) = 2T(n/2) + n$$

for $k \leq n$,

$$T(k) = 2T\left(\frac{k}{2}\right) + k.$$

$$T(n)$$



.....

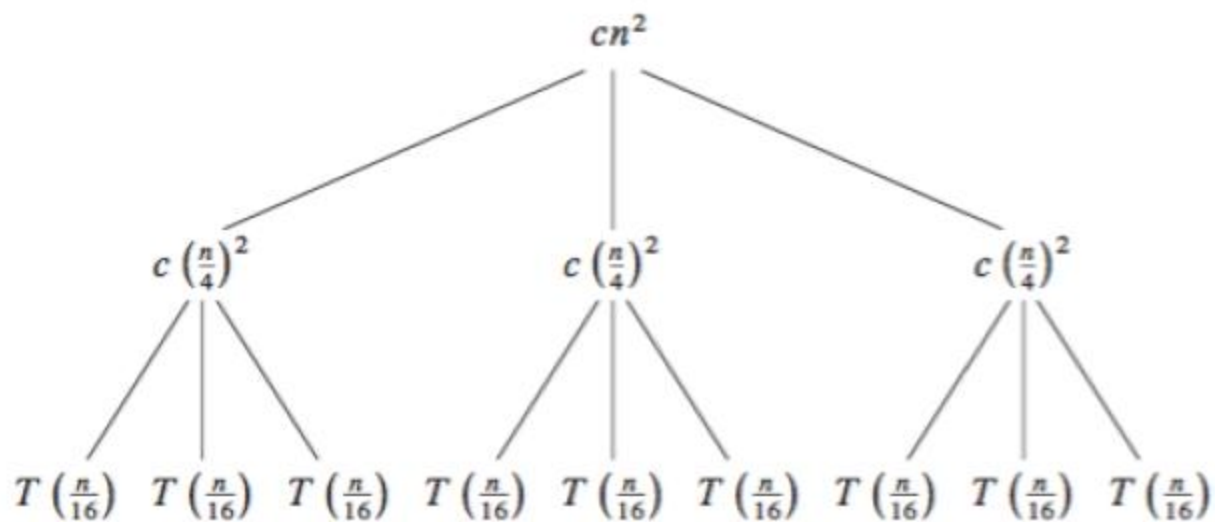
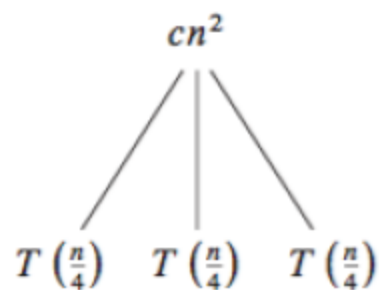
1 1 1 1

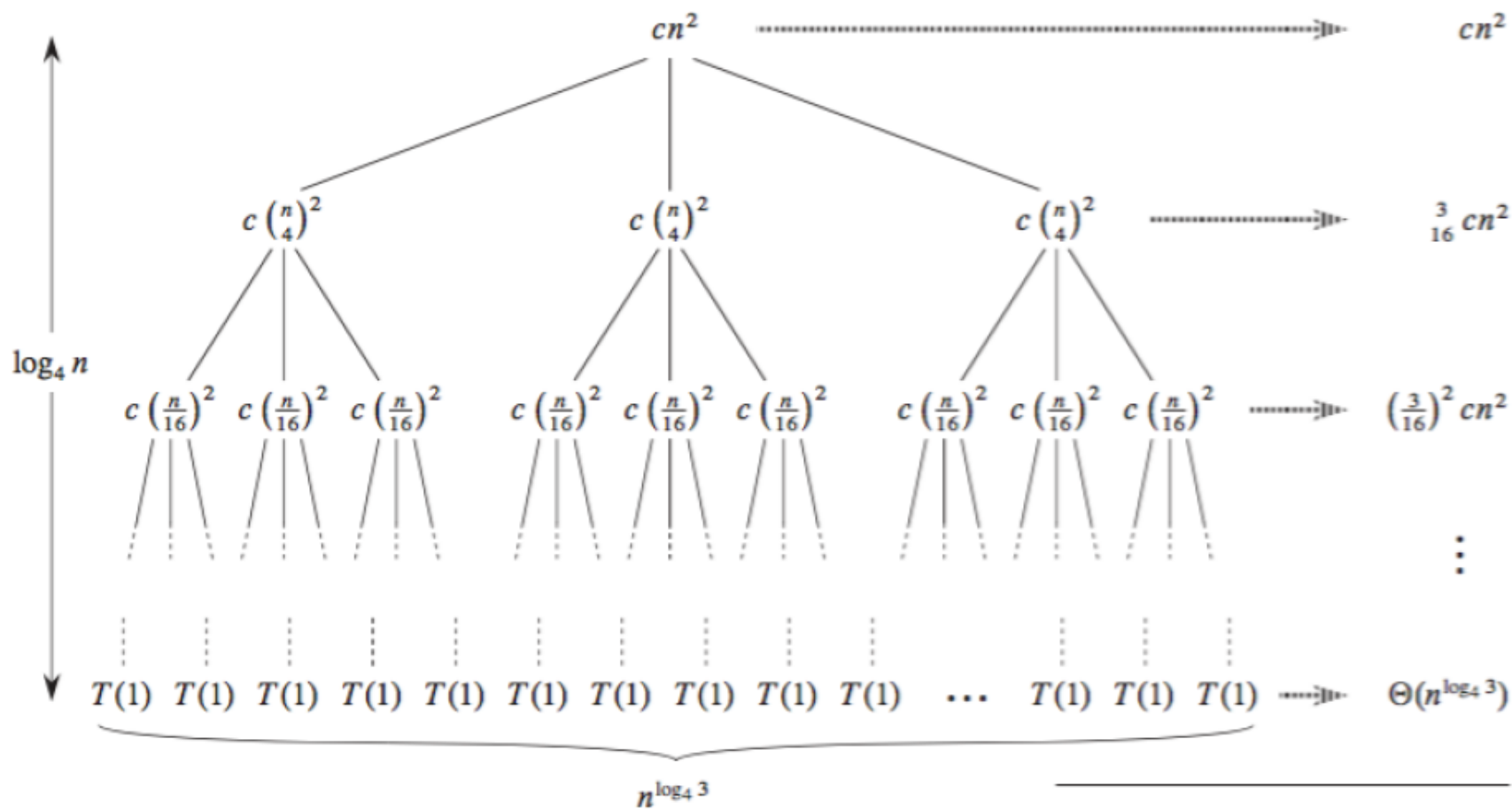
The sum of the values at each level of the tree is n (m nodes times the value $\frac{n}{m}$ in each node). There are $k = \log_2 n$ levels since we can cut the array in half k times. Thus, the sum of the elements in all nodes of the tree is $n \cdot \log_2 n$. Therefore, $T(n) = \Theta(n \cdot \log_2 n)$.



$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \quad \text{ie.} \quad T(n) = 3T(n/4) + cn^2.$$

$T(n)$





Total: $O(n^2)$

♦ Analysis: First we find the height of the recursion tree.
 Observe that a node at depth i reflects a subproblem of size $n/4^i$. The sub problem size hits $n = 1$ when $n/4^i = 1$ or $i = \log_4 n$. So the tree has $\log_4 n + 1$ levels.
 Now we determine the cost of each level of the tree. The number of nodes at depth i is 3^i . Each node at depth $i = 0, 1, \dots, \log_4 n - 1$ has a cost $c(n/4^i)^2$ of so the total cost of level i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. However, the bottom level is special. Each of the bottom nodes contribute cost $T(1)$, and there are $3^{\log_4 n} = n^{\log_4 3}$ of them.

So the total cost of the entire tree is

$$\begin{aligned}
 T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})
 \end{aligned}$$

- ◆ The left term is just the sum of a geometric series. So $T(n)$ evaluates to

$$\frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

- ◆ This looks complicated but we can bound it (from above) by the sum of the infinite series

$$\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3})$$

- ◆ Since functions in $\Theta(n^{\log_4 3})$ are also in $O(n^2)$, this whole expression is $O(n^2)$, Therefore, we can guess that $T(n) = O(n^2)$

Master Theorem

- ❖ This method is useful to solve the recurrences of the form, $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ and $f(n)$
- ❖ This recurrence describes an algorithm that divides a problem of size n into a subproblems, each of size n/b , and solves them recursively.

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

Simplified

◆ $T(n) = aT\left(\frac{n}{b}\right) + \theta(n^k \log^p n)$

◆ $a \geq 1, b > 1, k \geq 0$ and p is real number

⊕ Case 1: if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

⊕ Case 2: if $a = b^k$ then

a. If $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

b. If $p = -1$, then $T(n) = \theta(n^{\log_b a} \log \log n)$

c. If $p < -1$, then $T(n) = \theta(n^{\log_b a})$

⊕ Case 3: if $a < b^k$,

a. If $p \geq 0$ then $T(n) = \theta(n^k \log^p n)$

b. If $p < 0$ then $T(n) = O(n^k)$

Examples

1. $T(n) = 9T(n/3) + n$

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

$$f(n) = n$$

Comparing $n^{\log_b a}$ and $f(n)$

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

Satisfies Case 1 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

2. $T(n) = T(2n/3) + 1$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 = f(n)$$

Comparing $n^{\log_b a}$ and $f(n)$

$$n^{\log_b a} = \Theta(f(n))$$

Satisfies Case 2 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(\log n)$

$$3. T(n) = 2T(n/2) + n$$

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n = f(n)$$

Comparing $n^{\log_b a}$ and $f(n)$

$$n^{\log_b a} = \Theta(f(n))$$

Satisfies Case 2 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$

$$4. T(n) = 3T(n/4) + n \log n$$

$$n^{\log_b a} = n^{\log_4 3}$$

$$f(n) = n \log n$$

Comparing $n^{\log_b a}$ and $f(n)$

Satisfies Case 3 of Master's Theorem

Checking the regularity condition

$$a.f(n/b) < c.f(n) \text{ (for some constant } c < 1)$$

$$(3n/4) \log n/4 < c.n \log n$$

$$(3/4)n[\log n - \log 4] < (3/4)n \log n \text{ where } (c=3/4)$$

That implies the regularity condition is satisfied

That implies $T(n) = \Theta(f(n)) = \Theta(n \log n)$

