EECS660 **Topic 1: Algorithmic Fundamentals** **Kong**

Objective: To study the design & analysis of efficient computer algorithms for various computational problems.

Typical Approaches:

- Problems/Applications Driven
- Methods/Solutions Driven

Remark: We will use the Methods/Solutions Driven approach in EECS660.

Design:

Develop *correct and efficient algorithm* for solving various computational problems.

Analysis:

Proving the correctness and computing the (relative) efficiency of algorithms.

Topics Covered:

1. Basic Algorithmic Tools and Computation
2. Divide-and-Conquer
3. Dynamic Programming
4. Greedy Method
5. Intelligent Exhaustive Methods
6. Simple Bound Theory and Optimality of Algorithms
7. Graphs and Network Algorithms
8. Introduction to Problems Classification & NP-Completeness

Background Requirements:

Proficiencies in High-level Programming Language(s), Discrete Math, Probability & Data Structures.

Problem: A general question with a set of unspecified parameters.

Problem = {Question, Parameter(s)}

Instance of a problem: A problem with its set of parameters specified.

Example: Searching problem.
Input: Given n records R_1, R_2, …, R_n with keys k_1, k_2, …, k_n, $n \geq 1$, and a key x.
Output: Find index j such that record R_j will have key $k_j = x$. Return 0 if not found.

Parameters = {n, k_1, k_2, …, k_n, x}.

An instance of the searching problem:
Given 5 keys <18, 12, 26, 8, 15> with x = 8.

Another instance of the searching problem:
Given 6 keys <John, Mary, Alice, David, Wanda, Jim> with x = Emma.

Simplest Computational Problem:

Decision Problem: A problem with only yes- or no- answer.

Optimization Problems:

A problem with both a set of constraints and an objective function to be optimized by its solution.

Feasible Solution: Any solution satisfying the given constraints.
Optimal Solution: A feasible solution also optimizing the objective function.

Solving a problem:

To develop an "efficient" ***algorithm*** that can be used to generate a correct output for ***all*** possible instances (input) of the problem. An ***algorithm*** is a sequence of unambiguous step-by-step instructions that can be executed to solve a problem.

Q: How do we approach in solving a problem?

Basic Problems Solving Process:

1. *Problem Formulation*
2. *Algorithmic Development*
3. *Analysis of algorithm (correctness & efficiency)*
   Correctness of algorithm:
   Will the algorithm generate a correct output for each and every possible instance of the problem?
   Efficiency of algorithm:
   (i) How much computing resource will be needed in order to execute the algorithm?
   (ii) How do we design an "optimal" algorithm for solving a given problem?
   Most important factors: (CPU) *Time & Space* (Memory)
4. *Refinement*
5. *Program Development*
   Select/Design "good" data structures to implement the algorithm.
6. *Coding*
7. *Program Refinement, Verification & Debugging*

End Product:

An efficient program.

Factors Affecting the Efficiency of a Program:

1. Problem to be solved
2. Programming language used
3. Skill of programmers
4. Computing environment: Compiler and machine
5. Algorithm

Q: How do we describe an algorithm?

Method	**Simplicity**	**Precision**
English	Simplest	Least precise
Pseudo code		
High-Level P.L.	↓	↓
Low-Level P.L.		
Machine Language	Most complex	Most precise

Remark: *We will use English and pseudo code to describe our algorithms!*

Warning: *In describing your algorithm, you must first explain your algorithm in plain English, followed by your algorithm in pseudo code, then followed by coding if required.*

Format in expressing an algorithm:

Input:
Output:
Algorithm:
Correctness Analysis:
Complexity Analysis:

Q: How do we know that our algorithm is indeed correct?
How efficient is our algorithm?

Analysis of Algorithms:

Experimental Verification and Profiling vs. Analytical Approach

Experimental Verification/Profiling:

Implement the algorithm and then verify the correctness and measure the efficiency of the program with respect to set(s) of input data.

Major problem in using experimental profiling:

Highly machine/language/input dependent; results can be skewed if not careful! Also, you can't test your algorithm for all possible inputs.

Analytical Approach:

Prove the correctness of the algorithm using various proof techniques and mathematical analysis. For efficiency, identify a set of elementary (basic) operation(s) that will dominate the execution of the algorithm and then count it. Algorithms will be compared based on the number of these elementary operations used.

Major problem in using analytical approach:

Formal model and required computation can be quite complex; you must know your math well and remember how to count! ☺

Most important resource factors in performance analysis:

T(n) — *time complexity*
S(n) — *space complexity*

Resource increases as input size (n) increases!

Model of Computations: Random Access Machine (RAM).
Assumptions:

1. Only one instruction can be executed at a time.
   (Sequential machine)
2. Each datum is small enough to be stored in a single memory cell.
   (Uniform cost criterion)
3. Each stored datum can be accessed with the same constant cost.
   (Random access model)
4. Each basic operations such as read, write, +, -, ∗, /, compare(x,y), return, …, requires a constant cost.
   (Normalization)

Q: Let A be an algorithm for a given problem Π. What is the least, most, and average amount of computing resource required in order to execute A?

Algorithmic Fundamentals:

Let D_n be the set of all possible inputs to Π of size n,
C(I) be the amount of computing resource required to execute A with input I,
Pr(I) be the probability when I is the input to A, and
R(n) be the complexity function of A when executed with any input of size n.

Some Important Complexity Functions:

1. ***Best-Case Complexity:***
   $R_b(n) = \min_{I \in D_n} C(I)$
2. ***Worst-Case Complexity:***
   $R_w(n) = \max_{I \in D_n} C(I)$
3. ***Average-Case Complexity:***
   $R_a(n) = \Sigma_{I \in D_n} Pr(I) * C(I)$

Warning: In computing these complexity functions, do not choose a fixed value for n; they are all functions of n.

Remarks:

1. If there are k possible inputs of size n to Π and, if uniform distribution is assumed, then $\Pr(I) = \frac{1}{k}$. Hence,

$R_a(n) = (1/k)\Sigma_{I \in Dn} C(I)$, which is the "usual" way in computing the average of k quantities.

2. $R_a(n)$ is usually very difficult to compute. Different probability function will lead to different $R_a(n)$.

Recall that an algorithm is a sequence of step-by-step instructions.

Algorithm A:

S_1;

S_2;

•
•
•

S_m;

Q: How do we compute the complexity of A?

Compute the cost of each statement (instruction) and then sum them up.

Let $cost(S_i)$ be the cost in executing the statement S_i,
$1 \le i \le m$. Hence, $\mathbf{T(n) = \Sigma_{1 \le i \le m} cost(S_i).}$

Some Complications:

1. *S_i is a **conditional statement**:* if-then-else, case, switch, etc.
   $cost(S_i)$ = cost in evaluating the condition +
   cost in evaluating one of the branches
   Q: Which branch should we use?

2. *S_i is a **repetition (loop)**:* do-loop, while-loop, repeat-loop, etc.
   $cost(S_i)$ = (# times the loop condition is evaluated ∗
   cost in evaluating the loop condition) +
   (# times the loop is evaluated ∗
   cost in evaluating the body of the loop)
   Warning: *It can be very tricky in determining how many times a loop will be executed! Be careful.*

3. *S_i is a **recursive call**: direct and indirect recursions.*
   Need to set up and solve the recurrence equation for $cost(S_i)$.

Examples:

1. ***A simple searching problem***.

Input: An array A[1..n] of integers and an integer key x.

Output: Return the smallest integer i such that A[i] = x, $1 \le i \le n$, if exists; otherwise, return 0.

Approach:

Starting at A[1], sequentially compare x with A{1], A[2], …, A[n]. If there exist an i, $1 \le i \le n$, such that x = A[i], then return i; else return 0.

Sequential_Search algorithm:

Algorithm:

```
i = 1;
while i ≤ n and A[i] ≠ x do
    i = i + 1
endwhile;
if i ≤ n
    then return(i)
    else return(0)
endif;
```

Q: What are the operations involved in this algorithm?

Operations	**Cost**
assignment	c_1
comparison	c_2
logical-and	c_3
addition	c_4
return	c_5

Q: How do we compute the different complexity measures?

Let's look at the while-loop.

Q: How many times will the (body of) while-loop be executed?

min # times = 0 (x = A[1])
max # times = n (x = A[n] or $x \notin A$)
ave # times = ?

Detailed Analysis:

Best-case complexity:

$$T_b(n) = c_1 + (2c_2 + c_3) + (c_2 + c_5)$$
$$= c_1 + 3c_2 + c_3 + c_5$$

Worst-case complexity:

$$T_w(n) = c_1 + [(n+1)(2c_2 + c_3) + n(c_1 + c_4)] + (c_2 + c_5)$$

$= (c_1 + 2c_2 + c_3 + c_4)n + (c_1 + 3c_2 + c_3 + c_5)$

A Simplified Approach:

Instead of counting all operations, count the most important/dominant operation(s) only.

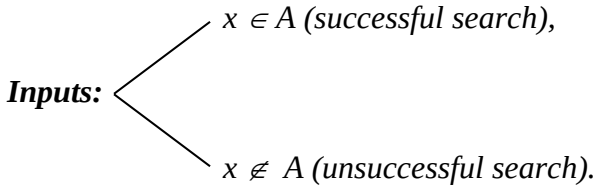
Let's now count the number of comparisons between x and A[i].

$T_b(n) = 1$
$T_w(n) = n$

Q: How about $T_a(n)$?

Recall that $T_a(n) = \Sigma_{I \in Dn} Pr(I) * C(I)$. Hence, we must first determine D_n, the set of all possible inputs of size n.

Observe that, for any searching problem, there are always two possible outcomes.



Based on these outcomes in searching, we must identify all possible inputs and their effects on the cost functions.

Q: If $x \in A$, where and for how many comparisons it will take in order to find x?

Location (input I), x = A[i]	x = A[1]	x = A[2]	…	x = A[i]	…	x = A[n]
# Comparisons, C(I)= C(x = A[i])	1	2		i		n

Q: What if $x \notin A$? Will different inputs affect the number of comparisons required to determine that $x \notin A$?

Whenever $x \notin A$, independent of x, our sequential algorithm must compare x with all of A[1], A[2], …, A[n] before it can be terminated. Hence, $C(x \notin A) = n$.

Q: What is Pr(x = A[i], $1 \le i \le n$) and Pr(x $\notin$ A)?
Need to know the distribution of inputs.

Let Pr(x $\in$A) = q, $0 \le q \le 1$. Assuming *uniform distribution*, we have

$$\Pr(x = A[i]) = \frac{q}{n},\ 1 \le i \le n, \text{ and } \Pr(x \notin A) = 1 - q.$$

Also, C(x =A[i]) = i, $1 \le i \le n$, and C(x $\notin$ A) = n.

Hence,

$$T_a(n) = \sum_{i=1}^{n} \frac{q}{n} * i + (1-q) * n = \frac{q(n+1)}{2} + (1-q)n.$$

Some Simple Cases:

1. q = 0: $T_a(n) = n$.

2. q = 1: $T_a(n) = \frac{n+1}{2}$.

3. q = 1/2: $T_a(n) = \frac{n+1}{4} + \frac{n}{2} = \frac{3n}{4} + \frac{1}{4}$.

Q: *What if A is sorted and we would still like to use the above sequential search algorithm for searching x?*

2. **Sequential Searching a Sorted Array:**

Input: A sorted array A[1..n] of n distinct integers and an integer key x.

Output: Return i, $1 \le i \le n$, if A[i] = x; otherwise, return 0.

Approach:

$\ldots < A[i-1] < A[i] < \ldots$

Observe that whenever $A[i-1] < x < A[i]$, we can terminate the search immediately.

Algorithm:

```
i = 1;
while i ≤ n and x > A[i] do
    i = i + 1
endwhile;
if i ≤ n
    then if x = A[i]
            then return i
            else return 0
         endif
    else return 0
endif;
```

Complexity Analysis:

Basic operation: Comparisons between x and A[i].

1. **Best-Case Complexity:**

   $T_b(n) = 2$

2. **Worst-Case Complexity:**

   $T_w(n) = n + 1$ (1 comparison each for loop condition/body)

3. **Average-Case Complexity:**

   Observe that, as in previous searching problem, there are always two possible outcomes.

Inputs: $x \in A$ *(successful search),*

$x \notin A$ *(unsuccessful search).*

For successful search, as in previous searching problem, there are n classes of inputs D_i with different cost functions, where D_i corresponds to the input when x = A[i], $1 \le i \le n$.

Q: For unsuccessful search, what kinds of inputs can we have and how will they affect the cost in searching?

Let's consider how our algorithm is terminated during an unsuccessful search.

x < A[1]	→	2 comparison	$x \in Gap_1$
A[1] < x < A[2]	→	3 comparisons	$x \in Gap_2$
A[2] < x < A[3]	→	4 comparisons	$x \in Gap_3$
…			…
A[n−1] < x < A[n]	→	n+1 comparisons	$x \in Gap_n$
x > A[n]	→	n+1 comparison	$x \in Gap_{n+1}$

Hence, there are n + 1 classes of inputs J_i, where J_i corresponds to the input with $x \in Gap_i$, $1 \le i \le n+1$.

Let $Pr(x \in A\} = q$; $0 \le q \le 1$. Assuming uniform distribution, we have

$$Pr(x = I_i) = \frac{q}{n} \text{ and } Pr(x = J_i) = \frac{1-q}{n+1}.$$

Hence,

$$\begin{aligned} & T_a(n) \\ &= \sum_{i=1}^{n} (\frac{q}{n}) * (i+1) + \sum_{i=1}^{n} (\frac{1-q}{n+1}) * (i+1) + (\frac{1-q}{n+1}) * (n+1) \\ &= \frac{q(n+1)}{2} + q + \frac{(1-q)n}{2} + \frac{n(1-q)}{n+1} + (1-q) \\ &\; \frac{qn+q+n-qn}{2} + (2-q) \\ &= \frac{n+q}{2} + (2-q). \end{aligned}$$

Some Simple Cases:

1. q = 0:
   $T_a(n)$
   $= \frac{n}{2} + 2$
   $\cong \frac{n}{2}.$

2. q = 1:
   $T_a(n)$
   $= \frac{n+1}{2} + 1$
   $\cong \frac{n}{2}.$

3. q = $\frac{1}{2}$:
   $T_a(n)$
   $= \frac{2n+1}{4} + \frac{3}{2}$
   $\cong \frac{n}{2}.$

3. Insertion Sort:
Input: An array A[1..n] of integers.
Output: Sorted array A in non-decreasing order, A[1] ≤ A[2] ≤ … ≤ A[n].

Approach: Insertion sort.
A[1] by itself is sorted.
For each element A[i], 2 ≤ i ≤ n, insert A[i] into its sorted position among A[1..i-1].

More precise algorithm:

```
if n = 1
    return;
for i = 2 to n do            // insert A[2], …, A[n]
    x = A[i];                // start to insert x
    A[0] = x;
    j = i-1;
    while x < A[j] do        // find position for x
       A[j+1] = A[j];        // shifting array elements
       j = j-1;
    endwhile;
    A[j+1] = x               // inset x into proper position
endfor;
```

Basic operation:
Comparisons between elements in A.

Best-case Complexity:
While-loop will not be executed *(array is sorted in non-decreasing order)*.

$$T_b(n) = \sum_{i=2}^{n} 1 = n - 1.$$

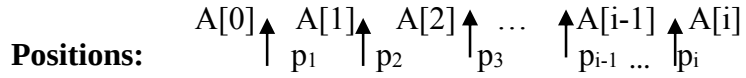
Worst-case Complexity:
While-loop will always be executed until A[0] is encountered (array is sorted in non-increasing order).

Q: How many comparisons are required to insert A[i]?

$$T_w(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1.$$

Average-case Complexity:

In inserting A[i] into its sorted position among elements A[0..i-1], there are i possible positions for A[i]:



Assumption:

Assume that x is equally likely to be at any of these i positions after insertion.

Hence, Pr(x at p_i) = $\frac{1}{i}$.

$$T_a(n)$$

$$= \sum_{i=2}^{n} \sum_{j=1}^{i} \frac{1}{i} * j$$

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

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

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

$$= \frac{1}{2} [\frac{n(n+1)}{2} + n] - 1$$

$$= \frac{n^2}{4} + \frac{3n}{4} - 1.$$

Recall that if an array A[1..n] is already sorted, we can search for a given key x in A by using binary search.

4. Binary Search:

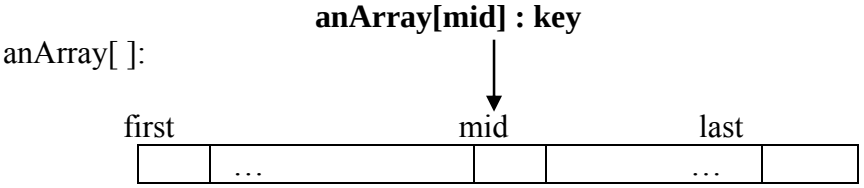
Input: A sorted array A[1..n] of n distinct integers and an integer key x.

Output: Return i, $1 \le i \le n$, if A[i] = x; otherwise, return 0.

Approach:

Prototype:

```
int bsearch(const int anArray[], int first, int last, int key);
```



key < anArray[mid] key > anArray[mid]

Recursive Algorithm:

```
mid = (first + last)/2;
if key = anArray[mid]
    return mid;
else if key < anArray[mid]
    bsearch(anArray, first, mid-1, key);
else                                          // if x > anArray[mid]
    bsearch(anArray, mid+1, last, key);
```

Recursive Binary Search Algorithm:

```
int bsearch(const int anArray[], int first, int last, int key)
{
    int index;
    if (first > last)          //  base case; key not found
        index = 0;
    else
    {
     int mid = (first + last)/2;          //          compute mid for dividing
     if (key == anArray[mid])             //    key found
     {
        index = mid;
     }
     else if (key < anArray [mid])        //          search left sub-array
                index = bsearch(anArray, first, mid – 1, key);
          else                                        //  search right sub-array
                index = bsearch(anArray, mid + 1, last, key);
    }
    return index;
}   //   end bsearch
```

Complexity Analysis:
Basic operation: Comparisons between x and A[i].

1. Best-Case Complexity:
   $T_b(n) = 1$

2. Worst-Case Complexity:
   $T(1) = 1,$

   $T(n) \le T(\frac{n}{2}) + 1, n > 1.$

   Assume that $n = 2^k$, $k \ge 1$. Using the Method of Repeated Substitutions, we have

   $T(n)$

   $\le T(\frac{n}{2}) + 1$

   $\le T(\frac{n}{2^2}) + 2$

   $\le T(\frac{n}{2^3}) + 3$

   ...

   $\le T(\frac{n}{2^k}) + k$

   $= \lg n + 1$

Observe that in all above examples T(n)'s are being represented by a very simple mathematical expression (elementary function). These are called the ***closed-form expression*** of T(n).

If T(n) = f(n), where f(n) is an elementary function, then T(n) can be computed exactly by substituting n into f(n).

Q: What if such an expression cannot be found (either doesn't exist or much too difficult to compute) to represent T(n)?

Use approximation!

We can simplify our computation by finding an elementary function f(n) such that $T(n) \le kf(n)$ for sufficiently large n.

Review: Asymptotic Analysis of Algorithms:
D**efn:** A function f: N $\rightarrow$ R is a ***positive function*** if f(n) > 0 for all n; f(n) is an ***eventually positive function*** if f(n) > 0 for all $n \geq n_0$.

Remark: Observe that all complexity functions are (eventually) positive functions. Also, unless specified otherwise, all functions considered in this course are eventually positive functions.

Defn: f(n) = O(g(n)) iff $\exists$ constants k > 0, $n_0 > 0$ such that $f(n) \leq k(g(n)) \ \forall \ n \geq n_0$.

Example: Prove that $\frac{n^3 - n^2 \lg n + 5n^2 - 10}{6n^2 - 7n^{3/2} + 3n - 8} = O(n)$.

$$\frac{n^3 - n^2 \lg n + 5n^2 - 10}{6n^2 - 7n^{3/2} + 3n - 8}$$
$$\leq \frac{n^3 + 5n^3}{6n^2 - 7n^{3/2} + 3n - 8}, n \geq 1$$
$$\leq \frac{6n^3}{6n^2 - 7n^{3/2} - 8}, n \geq 1$$
$$\leq \frac{6n^3}{2n^2 + (2n^2 - 7n^{3/2}) + (2n^2 - 8)}, n \geq 1$$
$$\leq \frac{6n^3}{2n^2}, n \geq 16$$
$$\leq 3n, n \geq 16.$$

By choosing k = 3 and $n_0 = 16$, we proved that $\frac{n^3 - n^2 \lg n + 5n^2 - 10}{6n^2 - 7n^{3/2} + 3n - 8} = O(n)$.

More Examples:
1. $2n^2 - 3n + 10 = O(n^2)$.
2. $2n^2 - 3n + 10 = O(n^k)$, $k \geq 2$.
3. $3\lg n! = O(n \lg n)$.
4. $n^2 - 3n^{16} + 2^n = O(2^n)$.
5. $n^2 - 36n \lg n - 1024 = O(n^2)$.
6. $n^2 - 36n \lg n - 1024 \neq O(n)$.
7. $2^{n+1} = O(2^n)$.
8. $4^n \neq O(3^n)$.

Defn: f(n) = Ω(g(n)) iff ∃ constants k > 0, $n_0 > 0$ such that f(n) ≥ k(g(n)) ∀ n ≥ n_0.

Theorem: f(n) = Ω(g(n)) iff g(n) = O(f(n)).

Example: Prove that $\frac{n^3 - n^2 \lg n + 5n^2 - 10}{6n^2 - 7n^{3/2} + 3n - 8} = \Omega(n)$.

$$\frac{n^3 - n^2 \lg n + 5n^2 - 10}{6n^2 - 7n^{3/2} + 3n - 8}$$

$$\geq \frac{n^3 - n^2 \lg n - 10}{6n^2 - 7n^{3/2} + 3n - 8}, n \geq 1$$

$$= \frac{\frac{n^3}{3} + (\frac{n^3}{3} - n^2 \lg n) + (\frac{n^3}{3} - 10)}{6n^2 - 7n^{3/2} + 3n - 8}, n \geq 1$$

$$\geq \frac{\frac{n^3}{3}}{6n^2 + 3n}, n \geq 2^5$$

$$\geq \frac{n^3}{3(6n^2 + 3n^2)}, n \geq 2^5$$

$$= \frac{1}{27} n, n \geq 2^5.$$

By choosing k = 1/27 and n_0 = 256, we proved that $\frac{n^3 - 8n^2 \lg n + 5n^2 - 10}{6n^2 - 7n^{3/2} + 3n - 8} = \Omega(n)$.

More Examples:

1. $2n^2 - 3n + 10 = \Omega(n^2)$.
2. $2n^2 - 3n + 10 \neq \Omega(n^3)$.
3. $3\lg n! = \Omega(n \lg n)$.
4. $n^2 - 3n^{16} + 2^n = \Omega(2^n)$.
5. $n^2 - 36n \lg n - 1024 = \Omega(n^2)$.
6. $n^2 - 36n \lg n - 1024 = \Omega(n)$.
7. $2^{n+1} = \Omega(2^n)$.
8. $4^n = \Omega(3^n)$.

Defn: $f(n) = \Theta(g(n))$ iff $\exists$ constants $k_1 > 0$, $k_2 > 0$, $n_0 > 0$ such that $k_2 g(n) \leq f(n) \leq k_1 g(n)$ $\forall$ $n \geq n_0$.

Theorem: The following statements are equivalent:

1. $f(n) = \Theta(g(n))$.
2. $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
3. $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Examples:

1. $2n^2 - 3n + 10 = \Theta(n^2)$.
2. $2n^2 - 3n + 10 \neq \Theta(n^3)$.
3. $3\lg n! = \Theta(n \lg n)$.
4. $n^2 - 3n^{16} + 2^n = \Theta(2^n)$.
5. $n^2 - 36n \lg n - 1024 = \Theta(n^2)$.
6. $n^2 - 36n \lg n - 1024 \neq \Theta(n)$.
7. $2^{n+1} = \Theta(2^n)$.
8. $4^n \neq \Theta(3^n)$.

Other Useful Asymptotic Notations:

Defn: $f(n) = o(g(n))$ iff $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$.
Defn: $f(n) = \omega(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) \neq \Theta(g(n))$.

Using Limits to Verify Asymptotic Behavior of a Function:

Limit Ratio Theorem: If $\lim_{n \to \infty} f(n)/g(n) = c$, then

1. if $0 \leq c < \infty$, then $f(n) = O(g(n))$,
2. if $0 < c \leq \infty$, then $f(n) = \Omega(g(n))$,
3. if $0 < c < \infty$, then $f(n) = \Theta(g(n))$,
4. if $c = 0$, then $f(n) = o(g(n))$,
5. if $c = \infty$, then $f(n) = \omega(g(n))$.

Warning: Even if $f(n) = \chi(g(n))$, $\lim_{n \to \infty} f(n)/g(n)$ may NOT exist, where χ is one of the asymptotic notations above.

Given an algorithm A.

Try to compute a function f(n) such that T(n) = f(n).

If not possible, try T(n) = Θ(f(n)).

If not possible, try T(n) = o(f(n)).

If not possible, try T(n) = O(f(n)).

Some Useful Function in Complexity Analysis:

f(n)	***Growth Rate***	***Algorithmic Performance***
n^n	Fastest	Worst
$n!$		
•		
•		
•		
3^n		
2^n	↑	↓
•		
•		
•		
$n^k, k \geq 2$		
n^2		
nlgn		
n		
lgn		
c	Slowest	Best (algorithm insensitive to n)

Some Useful Properties:

1. Reflexive property:
   For constant c,
   $cf(n) = O(cf(n)) = O(f(n))$.
   $cf(n) = \Omega(cf(n)) = \Omega(f(n))$.
   $cf(n) = \Theta(cf(n)) = \Theta(f(n))$.

2. Symmetric property:
   $f(n) \in \Theta(g(n))$ implies $g(n) \in \Theta(f(n))$.

3. Transitive property:
   If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
   If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.
   If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

4. Sum Rule:
   If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
   then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$.

   Remark: This sum rule can be extended to k functions, where k is a ***fixed*** integer constant.

5. If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n^1 + a_0$, where a_i's are constants with $a_m > 0$, then $f(n) = \Theta(n^m)$.

Examples: Prove or disprove the following statements:

1. If $T(n) = O(f(n))$ and $f(n) = O(n)$, then $T(n) = \Omega(n)$.
   Proof. **False.** Take $T(n) = 1$, $f(n) = n$.

2. If $T(n) = O(f(n))$ and $f(n) = \Omega(g(n))$, then $T(n) = O(g(n))$.
   Proof. **False.** Take $T(n) = n$, $f(n) = n^2$, $g(n) = 1$.

3. If $T(n) = \Theta(f(n))$ and $f(n) = \Theta(g(n))$, then $T(n) = g(n)$.
   Proof. **False.** Consider $T(n) = n^2$, $f(n) = 2n^2$, $g(n) = 3n^2$.

4. If $T(n) = \Theta(f(n))$ and $f(n) = \Omega(g(n))$, then $T(n) = \Omega(g(n))$.
   Proof. **True.** $T(n) = \Theta(f(n))$ implies $T(n) = \Omega(f(n))$. Since $f(n) = \Omega(g(n))$, by transitivity property, $T(n) = \Omega(g(n))$.

5. If $f(n) = O(n)$, then $[f(n)]^2 = O(n^2)$.
   Proof. **True.** $f(n) = O(n)$ implies that $f(n) \le kn$, for some positive constants k and n. Hence, $[f(n)]^2 \le k^2 n^2$ implying that $[f(n)]^2 = O(n^2)$.

6. If $f(n) = O(n)$, then $2^{f(n)} = O(2^n)$.
   Proof. **False.** Take $f(n) = 2n$. $2^{f(n)} = 2^{2n} = 4^n \neq O(2^n)$.

More Examples:

7. Prove that $f(n) = (n + 1)\lg(4n^2 + 60) = O(n\lg n)$

Observe that

$$\begin{aligned} \lg(4n^2 + 60) &\le \lg(4n^2 + 60n^2) \\ &= \lg(64n^2) \\ &= \lg(8n)^2 \\ &= 2\lg(8n) \\ &= 2(\lg 8 + \lg n) \\ &\le 2(\lg n + \lg n), \text{ for } n \ge 8 \\ &= 4\lg n. \end{aligned}$$

$$\begin{aligned} \therefore (n + 1)\lg(4n^2 + 60) &= O((n + 1)\lg(4n^2 + 60)) \\ &= O(n\lg(4n^2 + 60) + \lg(4n^2 + 60)) \\ &= O(\max\{n\lg(4n^2 + 60), \lg(4n^2 + 60)\}) \\ &= O(n\lg(4n^2 + 60) \\ &= O(4n\lg n \\ &= O(4n\lg n) \\ &= O(n\lg n) \end{aligned}$$

8. Given two algorithms A_1 and A_2 with $T_1(n) = 2^{18}n^2$ and $T_2(n) = 2^n$. Find smallest input size $n_0 > 0$ such that A_1 will perform faster than A_2.for all $n > n_0$.

Need to find smallest integer $n > 0$ such that $2^{18}n^2 \le 2^n$.

$$\begin{aligned} 2^{18}n^2 &\le 2^n \\ \lg 2^{18}n^2 &\le \lg 2^n \\ \lg 2^{18} + \lg n^2 &\le n \\ 18 + 2\lg n &\le n \\ 0 &\le n - 2\lg n - 18 \end{aligned}$$

Take $n = 2^4$, we have $2^4 - 2\lg 2^4 - 18 = -10$
Take $n = 2^5$, we have $2^5 - 2\lg 2^5 - 18 = 4$.

Hence, $2^4 < n < 2^5$.

Q: How do you find the smallest n that will satisfy the above inequality?
Apply binary search to the region $(2^4, 2^5)$.

Review: Some Important Summations.

1. $\Sigma_i (k_1 f(i) + k_2 g(i)) = k_1 \Sigma_i f(i) + k_2 \Sigma_i g(i)$, where k_1, k_2 are constants.

2. $\Sigma_{\alpha \le i \le \beta} k = (\beta - \alpha + 1)k$, where k is a constant and $\alpha \le \beta$ are integers.

3. $\Sigma_{\beta \le i \le \gamma} f(i) = \Sigma_{\alpha \le i \le \gamma} f(i) - \Sigma_{\alpha \le i \le \beta - 1} f(i)$, where $\alpha \le \beta \le \gamma$ are integers.

4. $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

5. $$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

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

7. $$\sum_{i=1}^{n} i^m = \frac{n^{m+1}}{(m+1)} + O(n^m).$$

8. $$\sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + O(\frac{1}{n}), \gamma = 0.577...$$

9. $$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r} \to \frac{1}{1 - r}, |r| < 1.$$

General (Simplified) Approach in Complexity Analysis using Dominating Step(s):
Approach:

1. Identify the dominating steps in the algorithm that will dominate and capture the performance function of the algorithm.
2. Assuming that all elementary operations will have the same constant cost.

Examples:

1.
```
x = 2;
y = 5;
for i = 1 to n do
        for j = 1 to n do
                for k = 1 to n do
                    y = x * y / 2;
                    x = x + y – 10;
                endfor;
        endfor;
endfor;
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} K$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{n} K n$$

$$= \sum_{i=1}^{n} K n^2 = K n^3.$$

2.
```
x = 5;
y = 60;
for i = 1 to n do
        for j = 1 to i do
                x = 2*x + 1;
        endfor;
        for k = 1 to n do
                y = x*y/2;
        endfor;
endfor;
```

T(n)

$$= \sum_{i=1}^{n} (\sum_{j=1}^{i} + \sum_{k=1}^{n}) C$$

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

$$= C [\frac{n(n+1)}{2} + n^2].$$

3.
```
k = 1;
x = 2;
y = 3;
while k <= n do
        for i = 1 to n do
                k = k + i;
        endfor;
        x = x + y;
endwhile;
```

Observe that the while-loop will only be executed once. Hence,

$$T(n) = \sum_{i=1}^{n} C = Cn.$$

4.
```
k = 1;
x = 6;
y = 60;
while k <= n^2 do
        x = (x*y + 2*x)/4;
        k = k*k;
endwhile;
```

Observe that k is always equal to 1. Hence, we have an infinite loop and T(n) = ∞.