Topic 1: Introduction to Algorithmic Analysis

Read: Chpt. 2, Weiss

Goals of EECS560:
To study the design, functionality, implementation, and performance of advance abstract data types (ADTs) for the development of “efficient” programs for solving real-world computational problems.

Solving a Problem:
To develop an efficient algorithm/program that can be used to generate a correct output for all possible inputs to the problem.

Algorithm:
An algorithm is a sequence of step-by-step instructions/statements that can be executed to solve a problem in a finite amount of time.

Algorithm:
\begin{align*}
S_1; \\
S_2; \\
\ldots \\
\ldots \\
\ldots \\
S_m; \quad S_i - \text{statement.}
\end{align*}
Statements:

- **Simple**: assign, compare, return, add, …
- **Structured**: switch, loop, function call, …

**Characteristics of an Algorithm:**
- Read by human
- Machine/language independent
- Unambiguous
- Must terminate

**Remark:** Algorithm operates on data.

**Data structure:** Collection of data objects organized in some manner so as to facilitate computation in an efficient manner.

**Program = Algorithm + Data Structures**

**Algorithm vs. Program:**

<table>
<thead>
<tr>
<th>Program</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine/Language dependent</td>
<td>Independent</td>
</tr>
<tr>
<td>May not terminate</td>
<td>Must terminate</td>
</tr>
</tbody>
</table>

**Remark:** In studying data structure designs, we will first concentrate on abstract data types (ADTs).
Abstract Data Type: A mathematical system consisting of a collection of data objects together with some operations (algorithms) defined on them.

Remark: EECS560 is not a course about C++, Java, or any other programming language; it is a course on data structures development. We will concentrate on the design and analysis of different classes of Abstract Data Types (ADT) and their implementations. In EECS660, we will concentrate on the design and analysis of algorithms and use the ADTs developed here for solving various computational problems.

Algorithm Specification:

<table>
<thead>
<tr>
<th>Method</th>
<th>Simplicity</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>Simplest</td>
<td>Least precise</td>
</tr>
<tr>
<td>Pseudo code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-Level P.L.</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>Low-Level P.L.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Language</td>
<td>Most complex</td>
<td>Most precise</td>
</tr>
</tbody>
</table>

Remark: In describing your algorithm, you must first explain your approach in plain English, followed by your algorithm in pseudo code, then C++ code if required.
Format in expressing an algorithm:

*Input:*

*Output:*

*Algorithm:*

*Correctness Analysis:*

*Complexity Analysis:*

Example: A simple searching problem.

*Input:* An array $A[1..n]$ of $n$ distinct integers, $n \geq 1$, and an integer key $x$.

*Output:* Return integer $i$, $1 \leq i \leq n$, if $A[i] = x$. Else, return 0.

*Approach:* Sequential search algorithm.

$$
\begin{array}{cccccccc}
\downarrow & \downarrow & \downarrow & & \downarrow \\
x & \ldots & & & \\
\end{array}
$$
Algorithm:
Sequential_Search(A:array, x: integer);
    i = 1;
    while i ≤ n and A[i] ≠ x do // find x sequentially
        i = i + 1
    endwhile;
    if i ≤ n
        then return(i) // A[i] = x found
        else return(0) // x ∉ A
    endif;
end Sequential_Search;

Q: How good/bad is this algorithm? More importantly, how do we know that this algorithm is indeed correct?

Analysis of Algorithms:
Correctness and efficiency of Algorithm.
Experimental vs. Analytical Approaches:

Experimental approach:
Implement a given algorithm in a programming language and then execute the program in a machine with a suitable set of input data. Verify the correctness of the algorithm/program and measure the CPU time/memory used.


Analytical approach:
Prove the correctness of the algorithm using various proof techniques. Identify some elementary operations in the algorithm that will dominate the execution time complexity of the algorithm and then count them. Memory/space complexity can be computed in a similar fashion.

Problems: Mathematically involved; can be extremely difficult. EECS258/268: Concentrate on experimental approach. EECS560/660: Concentrate on analytical approach.

Q: How do we measure the “goodness” of an algorithm?

Most important cost factors:

\[ T(n) \] — time complexity
\[ S(n) \] — space complexity
Let A be an algorithm for a given problem \( \Pi \). 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 \( \Pi \) 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,
\( R(n) \) be the complexity function of A when executed with any input of size \( n \).

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:**
   \( R_a(n) = \sum_{I \in D_n} \Pr(I) \cdot C(I) \)

**Remark:**
\( R_a(n) \) is usually very difficult to compute. Different probability functions will lead to different \( R_a(n) \).
Model of Computation:  
*Random Access Machine* (RAM):
  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)*

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

Algorithm A:

\[
S_1; \\
S_2; \\
\cdot \\
\cdot \\
\cdot \\
S_m; 
\]

Let \( \text{cost}(S_i) \) be the cost in executing the statement \( S_i \), \( 1 \leq i \leq m \),

\[
T(n) = \Sigma_{1 \leq i \leq m} \text{cost}(S_i).
\]
Some Complications:
1. $S_i$ is a **conditional statement**: if-then-else, case, switch, etc.
   
   \[
   \text{cost}(S_i) = \text{cost in evaluating the condition} + \\
   \text{cost in evaluating one of the branches}
   \]

   **Q:** Which branch should we execute? How do we compute $R_b(n)$, $R_w(n)$, or $R_a(n)$?

2. $S_i$ is a **repetition (loop)**: do-loop, while-loop, repeat-loop, etc.

   \[
   \text{cost}(S_i) = (\text{# times the loop condition is evaluated} \times \\
   \text{cost in evaluating the loop condition}) + \\
   (\text{# times the loop is evaluated} \times \\
   \text{cost in evaluating the body of the loop})
   \]

   **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**:
   
   May need to set up and solve the recurrence equation for $\text{cost}(S_i)$. 
Let’s now compute $T(n)$ for the Sequential Search algorithm.

1. Detailed Analysis:
   Approach: Assign a constant cost to each and every operations and then compute the total cost in executing the algorithm.

   Three statements with the following operations and costs:
   
<table>
<thead>
<tr>
<th>Operations</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>$c_1$</td>
</tr>
<tr>
<td>comparison</td>
<td>$c_2$</td>
</tr>
<tr>
<td>logical-and</td>
<td>$c_3$</td>
</tr>
<tr>
<td>addition</td>
<td>$c_4$</td>
</tr>
<tr>
<td>return</td>
<td>$c_5$</td>
</tr>
</tbody>
</table>

   Q: How many times will the while-loop be executed?
   
   min # times = 0 \quad (x = A[1])
   
   max # times = n \quad (x = A[n] \text{ or } x \not\in A)
   
   average # times = ?

   **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:
\[ T_w(n) = c_1 + [(n+1)(2c_2 + c_3) + n(c_1 + c_4)] + (c_2 + c_5) = (c_1 + 2c_2 + c_3 + c_4)n + (c_1 + 3c_2 + c_3 + c_5) \]

2. Simplified Approach Based On Dominating steps:
Approach: We first combine all related steps with constant cost and assign them a constant cost. Next, identify steps that will dominate the execution of the algorithm and compute the total cost in executing these dominating steps. If only few basic operations are involved in these steps, we may further simplify the computation by assuming that all these basic operations in these steps have the same constant cost.

Dominating step in sequential search algorithm:
\begin{verbatim}
while i \leq n and A[i] \neq x do // find x
    i = i + 1
endwhile;
\end{verbatim}

Best-case complexity:
\[ T_b(n) = C. \]

Worst-case complexity:
\[ T_w(n) = \sum_{i=1}^{n} C = Cn. \]
More Examples: Computing $T(n)$.

1. $x = 2$;
   $y = 5$;
   for $i = 1$ to $n$ do
     for $j = 1$ to $n$ do
       for $k = 1$ to $n$ do
         $y = x \times y / 2$;
         $x = x + y - 10$;
       endfor;
     endfor;
   endfor;

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

\[
= \sum_{i=1}^{n} \sum_{j=1}^{n} Kn
\]

\[
= \sum_{i=1}^{n} Kn^2 = Kn^3.
\]
2. \( x = 5; \)
\[
\begin{align*}
  y &= 60; \\
  &\text{for } i = 1 \text{ to } n \text{ do} \\
  &\quad \text{for } j = 1 \text{ to } i \text{ do} \\
  &\quad\quad \text{x} = 2\times x + 1; \\
  &\quad \quad \text{endfor;} \\
  &\quad \text{endfor;} \\
  &\quad \text{for } k = 1 \text{ to } n \text{ do} \\
  &\quad\quad \text{y} = x\times y/2; \\
  &\quad \quad \text{endfor;} \\
  &\quad \text{endfor;} \\
  \end{align*}
\]

\[T(n)\]
\[
= \sum_{i=1}^{n} \left( \sum_{j=1}^{i} + \sum_{k=1}^{n} \right) C \\
= C \sum_{i=1}^{n} (i + n) \\
= C \left[ \frac{n(n + 1)}{2} + n^2 \right].
\]
3. \( k = 1; \)
   \( x = 2; \)
   \( y = 3; \)
   while \( k \leq n \) 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 \( k \leq n^2 \) 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 \( T(n) = \infty. \)
3. Simplified Approach Based On Basic Operations:

Approach: Instead of counting all operations, identify most important basic operations and count them.

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

\[ T_b(n) = 1 \]
\[ T_w(n) = n \]

Q: How about \( T_a(n) \)?

Recall that \( T_a(n) = \sum_{I \in D_n} Pr(I) \cdot C(I) \).

*Must* first determine \( D_n \), the set of all possible inputs of size \( n \).

Inputs:

- \( x \in A \) (successful search),
- \( x \not\in A \) (unsuccessful search).

Q: If \( x \in A \), where can we find \( x \)?

- \( x \in A[1] \), \( x \in A[2] \), ..., or \( x \in A[n] \).

Q: What is \( Pr(x \in A[i], 1 \leq i \leq n) \)?
Let $\Pr(x \in A) = q$, by assuming uniform distribution, we have

$$\Pr(x \in A[i]) = \frac{q}{n}, 1 \leq i \leq n,$$

and

$$C(x \in A[i]) = i, 1 \leq i \leq n.$$

Also, $\Pr(x \not\in A) = 1 - q$, and $C(x \not\in A) = n$.

Hence,

$$T_a(n) = \sum_{I \in D_n} \Pr(I) \cdot C(I)$$

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

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

**Some Simple Cases:**

1. $q = 0$:

   $$T_a(n) = n.$$

2. $q = 1$:

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

3. $q = \frac{1}{2}$:

   $$T_a(n) = \frac{n + 1}{4} + \frac{n}{2} = \frac{3n}{4} + \frac{1}{4}. $$
From all previous examples, observe that all T(n)’s are being represented by a very simple mathematical expression (elementary function). These are called the \textit{closed-form expression} of T(n).

\textbf{Remark:} 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).

\textbf{Q:} What if such an expression can’t be found (either doesn’t exist or much too difficult to compute) to represent T(n)?

\textit{Use approximation!}

In order to provide a guarantee on how much computing resource is needed in executing A, we may want to find an elementary function f(n) such that T(n) \leq f(n) for all n.

We can simplify our computation even further by finding an elementary function f(n) such that T(n) \leq kf(n) for sufficiently large n.
Review: Asymptotic Analysis of Algorithms

Defn: A function f: N → 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 ≥ n₀.

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 ∃ constants k > 0, n₀ > 0 such that f(n) ≤ k(g(n)) ∀ n ≥ n₀.

Examples:
1. 2n² − 3n + 10 = O(n²).
2. 2n² − 3n + 10 ≠ O(n³).
3. 3lgn! = O(nlgn).
4. n² − 3n¹⁶ + 2ⁿ = O(2ⁿ).
5. n² − 36nlgn − 1024 = O(n²).
6. n² − 36nlgn − 1024 ≠ O(n).
7. 2ⁿ⁺¹ = O(2ⁿ).
8. 4ⁿ ≠ O(3ⁿ).
**Defn:** \( f(n) = \Omega(g(n)) \) iff \( \exists \) constants \( k > 0, n_0 > 0 \) such that
\( f(n) \geq k(g(n)) \ \forall \ n \geq n_0. \)

**Theorem:** \( f(n) = \Omega(g(n)) \) iff \( g(n) = O(f(n)). \)

**Examples:**
1. \( 2n^2 - 3n + 10 = \Omega(n^2). \)
2. \( 2n^2 - 3n + 10 \neq \Omega(n^3). \)
3. \( 3\text{lgn!} = \Omega(n\text{lgn}). \)
4. \( n^2 - 3n^{16} + 2^n = \Omega(2^n). \)
5. \( n^2 - 36n\text{lgn} - 1024 = \Omega(n^2). \)
6. \( n^2 - 36n\text{lgn} - 1024 = \Omega(n). \)
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 equivalence:
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. \( 3\ln n! = \Theta(n\ln n). \)
4. \( n^2 - 3n^{16} + 2^n = \Theta(2^n). \)
5. \( n^2 - 36n\ln n - 1024 = \Theta(n^2). \)
6. \( n^2 - 36n\ln n - 1024 \neq \Theta(n). \)
7. \( 2^{n+1} = \Theta(2^n). \)
8. \( 4^n \neq \Theta(3^n). \)
### Some Useful Function in Complexity Analysis:

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>Growth Rate</th>
<th>Algorithmic Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^n$</td>
<td>Fastest</td>
<td>Worst</td>
</tr>
<tr>
<td>$n!$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3^n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>$n^k$, $k \geq 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n\log n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\log n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c$</td>
<td>Slowest</td>
<td>Best</td>
</tr>
</tbody>
</table>

**HW:** Review logarithmic and exponential functions.
Given an algorithm A.
Try to compute a function \( f(n) \) such that \( T(n) = f(n) \).
   If not possible, try \( T(n) = \Theta(f(n)) \).
   If not possible, try \( T(n) = o(f(n)) \).
   If not possible, try \( T(n) = O(f(n)) \).

Some Important Properties of big-O, big-\( \Omega \), and big-\( \Theta \):
1. **Reflexive property:**
   \( f(n) = O(f(n)) \),
   \( f(n) = \Omega(f(n)) \),
   \( f(n) = \Theta(f(n)) \).

2. **Symmetric property:**
   \( f(n) = \Theta(g(n)) \) implies \( g(n) = \Theta(f(n)) \).

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 \( f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\}) \).
If \( f_1(n) = \Omega(g_1(n)) \) and \( f_2(n) = \Omega(g_2(n)) \),
then \( f_1(n) + f_2(n) = \Omega(\min\{g_1(n), g_2(n)\}) \).

If \( f_1(n) = \Theta(g_1(n)) \) and \( f_2(n) = \Theta(g_2(n)) \),
then \( f_1(n) + f_2(n) = \Theta(g_1(n)) \) and \( f_1(n) + f_2(n) = \Theta(g_2(n)) \).

5. **Product Rule:**
Given two (positive) functions \( g_1(n) > 0, g_2(n) > 0, \forall n \geq n_0 \).

If \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \),
then \( f_1(n) \ast f_2(n) = O(g_1(n)g_2(n)) \).

If \( f_1(n) = \Omega(g_1(n)) \) and \( f_2(n) = \Omega(g_2(n)) \),
then \( f_1(n) \ast f_2(n) = \Omega(g_1(n)g_2(n)) \).

If \( f_1(n) = \Theta(g_1(n)) \) and \( f_2(n) = \Theta(g_2(n)) \),
then \( f_1(n) \ast f_2(n) = \Theta(g_1(n)g_2(n)) \).

**Remark:** The sum rule and product rule in (5) & (6) can be extended to \( k \) functions, where \( k \) is a fixed integer constant.

6. 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 \( f(n) = \Theta(n^m) \).
Other 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:
**Theorem:** If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \), 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 \( f(n) = \chi(g(n)) \), \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \) may NOT exist, where \( \chi \) is one of the asymptotic notations above.
Review of some important summation equations:
1. \( \sum_i (k_1f(i) + k_2g(i)) = k_1\sum_i f(i) + k_2\sum_i g(i) \), where \( k_1, k_2 \) are constants.

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

3. \( \sum_{\beta \leq i \leq \gamma} f(i) = \sum_{\alpha \leq i \leq \gamma} f(i) - \sum_{\alpha \leq i \leq \beta - 1} f(i) \), where \( \alpha \leq \beta \leq \gamma \) are integers.

4. \( \sum_{1 \leq i \leq n} i = n(n+1)/2 \).

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

6. \( \sum_{1 \leq i \leq n} i^3 = [n(n+1)^2]/4 \).

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

8. \( \sum_{1 \leq i \leq n} 1/i = \ln n + \gamma + O(1/n) \), \( \gamma = 0.577 \ldots \) is the Euler’s constant.

9. \( \sum_{0 \leq i \leq n} r^i = (1-r^{n+1})/(1-r) \).

This geometric series converges to \( 1/(1-r) \) if \(|r| < 1\).
More Examples:
1. \( f(n) = 3n^2 + n\log n - 100n \)
   \[ = O(3n^2 + n\log n - 100n) \]
   \[ = O(\max\{3n^2, n\log n, -100n\}) \]
   \[ = O(3n^2) \]
   \[ = O(n^2) \]

2. \( f(n) = (n + 1)\log(4n^2 + 60) \)
   
   Observe that
   \[ \log(4n^2 + 60) \leq \log(4n^2 + 60n^2) \]
   \[ = \log(64n^2) \]
   \[ = \log(8n)^2 \]
   \[ = 2\log(8n) \]
   \[ = 2(\log 8 + \log n) \]
   \[ \leq 2(\log n + \log n), \text{ for } n \geq 8 \]
   \[ = 4\log n. \]

   \[ f(n) = (n + 1)\log(4n^2 + 60) \]
   \[ \leq (n + 1)(4\log n) \]
   \[ = 4n\log n + 4\log n \]
   \[ = O(\max\{4n\log n, 4\log n\}) \]
   \[ = O(4n\log n) \]
   \[ = O(n\log n) \]
3. 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 $n$ such that $A_1$ is faster than $A_2$.

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

$$
\begin{align*}
2^{18}n^2 & \leq 2^n \\
\lg2^{18}n^2 & \leq \lg2^n \\
\lg2^{18} + \lg n^2 & \leq n \\
18 + 2\lg n & \leq n \\
0 & \leq n - 2\lg n - 18
\end{align*}
$$

Take $n = 2^4$, we have $2^4 - 2\lg2^4 - 18 = -10$
Take $n = 2^5$, we have $2^5 - 2\lg2^5 - 18 = 4$.

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 $(2^4, 2^5)$.*