07-16 Class Session: Asymptotic Analysis and Algorithm Optimization

> Date & Time: 2025-07-16 09:08:54
> Location: [Insert Location]
> Course Name: [Enter Course Name]

Keywords

`Asymptotic Analysis` `Algorithm Complexity` `Fibonacci Sequence`

Key Learnings

1. Asymptotic Analysis: Asymptotic analysis is a method used in the context of algorithms to evaluate the efficiency of an algorithm by analyzing its runtime as the input size grows. It helps determine if an algorithm is ‘good’ by comparing its speed to other algorithms, focusing on how fast or slow it is relative to others.
2. Fibonacci Numbers and Recursive Algorithms: Fibonacci numbers are a sequence defined by a specific recurrence relation, often used as an example to illustrate recursion in algorithms. The sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding numbers. Recursive algorithms can be used to compute Fibonacci numbers by solving smaller subproblems.
3. Algorithm Implementation and Analysis: Recursive Fibonacci: Implementation of the Fibonacci sequence using a recursive Java method, and analysis of its time complexity and efficiency.
4. Algorithm Optimization: Dynamic Programming for Fibonacci: Optimization of the Fibonacci algorithm using dynamic programming (bottom-up approach) to improve time complexity from exponential to linear.
5. Input Size Measurement and Its Impact on Algorithm Analysis: Discussion on how to properly measure the size of an algorithm’s input, especially for numerical problems, and how this affects the interpretation of running time.
6. Output Size Considerations in Algorithm Analysis: Analysis of the relationship between input size (number of bits in n) and output size (number of bits in Fibonacci(n)), and its implications for algorithm efficiency.
7. Growth of Fibonacci Numbers and Bit Representation: The Fibonacci sequence grows exponentially, and the number of bits required to represent Fibonacci(n)

increases rapidly with n. The relationship between the input size (in bits) and the output size (in bits) is crucial for understanding algorithm complexity.

8. Measuring Algorithm Complexity by Input and Output Bits: Algorithm complexity should be measured in terms of the number of bits in the input and output, not just the value of n. For many algorithms, the number of bits per element is constant, but for others (like Fibonacci), the output size grows much faster than the input.
9. Prime Number Checking Algorithm and Complexity: The naive algorithm for checking if a number n is prime involves checking divisibility from 2 up to n-1. The input is given in log2(n) bits, but the algorithm may require up to n steps, which is exponential in the number of input bits.
10. Prime Number Checking and Square Root Optimization: Explains why checking divisibility up to the square root of n is sufficient for determining if n is prime, and the implications for algorithmic complexity.
11. Algorithmic Complexity Relative to Input Size in Bits: Discusses how the running time of algorithms, especially for prime checking, scales with the number of bits in the input, and why this matters for cryptography.
12. Importance of Prime Checking in Cryptography: Explains the role of prime number checking and factorization in public key cryptography, such as SSH and SSL, and why the difficulty of factoring large numbers underpins security.
13. Algorithm Output Size and Its Impact on Running Time: Compares the output size requirements for different problems (prime checking vs. Fibonacci number calculation) and how this affects the minimum possible running time.
14. Measuring Algorithm Complexity: Time and Space: Discusses how to measure the complexity of algorithms in terms of both time and space, and the importance of considering both input size and program size.
15. Approximating Algorithm Running Time: Introduces the concept of using functions like t(n) to represent the exact running time of a program and the need to approximate this function for analysis.
16. Big O, Omega, and Theta Notations: Big O, Omega, and Theta notations are mathematical tools used to describe the upper bound, lower bound, and tight bound of the running time of algorithms with respect to input size. These notations are sets of functions, not individual functions, and are used to approximate or bound the running time of algorithms.

Explanations

1. Asymptotic Analysis

- **Key Points**
  - Asymptotic analysis is used to understand the runtime of an algorithm.
  - It assumes the algorithm works correctly and produces the right output.

- The analysis focuses on how much time an algorithm takes for a certain input size, denoted as n.
- Algorithms are considered better if they are faster compared to others.
- The goal is to determine what makes an algorithm fast or slow.

- **Explanation**
  [Speaker 1] introduces asymptotic analysis as a way to evaluate algorithms beyond just correctness, focusing on runtime. The process involves considering the input size (n) and comparing the speed of different algorithms. The class is encouraged to think about what makes an algorithm fast or slow, setting the stage for deeper analysis.

2. Fibonacci Numbers and Recursive Algorithms

- **Key Points**
  - Fibonacci numbers are defined as: Fibonacci(0) = 0, Fibonacci(1) = 1, and for n > 2, Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2).
  - The sequence begins: 0, 1, 1, …
  - Fibonacci numbers appear in various natural phenomena.
  - Recursive algorithms solve problems by breaking them into smaller subproblems.
  - A recursive solution for Fibonacci uses the answers for n-1 and n-2 to compute n.
- **Explanation**
  [Speaker 1] and [Speaker 2] discuss the definition of Fibonacci numbers, emphasizing the recurrence relation and its natural occurrence. The class is shown how recursion can be used to solve the Fibonacci problem by leveraging solutions to smaller instances. Pseudocode is suggested, following the mathematical definition directly, and the importance of understanding recursion is highlighted.
- **Recursive Implementation of Fibonacci Numbers**
  > If n equals 0, return 0. If n equals 1, return 1. Otherwise, return Fibonacci(n-1) + Fibonacci(n-2).

  i. The function checks if n is 0 or 1, returning the base case values.
  ii. For n greater than 1, it recursively calls itself for n-1 and n-2.
  iii. This directly translates the mathematical definition into code.
  iv. The approach demonstrates how recursion can solve problems by reducing them to smaller, similar subproblems.

3. Algorithm Implementation and Analysis: Recursive Fibonacci

- **Key Points**
  - A recursive method is used to compute Fibonacci numbers.

    - The time to compute the nth Fibonacci number is measured.
    - Discussion on how the running time depends on the input n.
    - Comparison of possible time complexities: proportional to n, n squared, n factorial, or 2 to the n.
    - Fibonacci of n is smaller than 2 to the n, but greater than 2 to the power of n over 2.
    - The time to compute Fibonacci of n (T(n)) is proportional to the value of the nth Fibonacci number.
    - Empirical measurement and plotting of running time versus n.
    - Observation that running time increases rapidly as n increases.
- **Explanation**
  The class begins by implementing the Fibonacci sequence recursively in Java. The instructor discusses the importance of not just finding a solution, but finding the most efficient one. The time to compute the nth Fibonacci number is measured for values up to 30, and students are asked to estimate the time complexity. The instructor explains that the recursive approach leads to a running time proportional to the value of the nth Fibonacci number, which grows rapidly. The relationship T(n) = T(n-1) + T(n-2) is established, mirroring the Fibonacci recurrence. Empirical results show that the running time is proportional to the Fibonacci number itself, which is less than 2^n but greater than 2^(n/2).
- **Measuring Recursive Fibonacci Running Time**

  > The instructor implements a recursive Java method to compute Fibonacci numbers. For n from 1 to 30, the time taken to compute each Fibonacci number is measured and plotted. The running time is compared to the value of the Fibonacci number and to 2^n.

    i. Implement the recursive Fibonacci method in Java.
    ii. For each n from 1 to 30, measure the time taken to compute Fibonacci(n).
    iii. Plot the running time and compare it to the value of Fibonacci(n) and 2^n.
    iv. Observe that the running time increases rapidly and is proportional to the Fibonacci number.

4. Algorithm Optimization: Dynamic Programming for Fibonacci

- **Key Points**
    - Recognition of redundant computation in the recursive approach.
    - Introduction of memoization or tabulation to store previously computed values.
    - Implementation of a bottom-up approach using an array to store Fibonacci numbers.
    - Initialization of the array with base cases: index 0 = 0, index 1 = 1.
    - Computation of Fibonacci numbers from 2 to n using a loop.

    - Time complexity is reduced to O(n), with n additions.
    - Need to use an array of longs instead of ints to avoid integer overflow for large n.
    - Empirical measurement shows running time is now proportional to n, and much faster than the recursive approach.
- **Explanation**
  The instructor identifies that the recursive approach recomputes the same Fibonacci values multiple times. By storing previously computed values in an array (dynamic programming), redundant work is eliminated. The bottom-up approach initializes the first two values and iteratively computes each subsequent Fibonacci number. The running time is now linear in n, a significant improvement over the exponential time of the recursive method. The instructor demonstrates the implementation and shows that the running time is now a straight line when plotted against n. An issue with integer overflow is encountered and resolved by switching from int to long.
- **Dynamic Programming Implementation for Fibonacci**
  > If the input n is less than or equal to 1, return n. Otherwise, initialize an array of size n+1. Set array[0] = 0 and array[1] = 1. For i from 2 to n, set array[i] = array[i-1] + array[i-2]. Return array[n]. When using int for the array, an underflow occurs for large n, so the array type is changed to long.

    i. Identify that recursive calls recompute Fibonacci(n-2) multiple times.
    ii. Implement an array to store computed Fibonacci values.
    iii. Initialize base cases and use a loop to fill the array.
    iv. Measure running time and observe linear growth.
    v. Switch array type to long to handle large Fibonacci numbers.

5. Input Size Measurement and Its Impact on Algorithm Analysis

- **Key Points**
    - Input size is often measured by the number of elements (n) for arrays, but for numerical inputs, it should be measured by the number of bits required to represent the input.
    - For example, the number 30 requires 2 digits, 32 requires 2 digits, 300 requires 3 digits.
    - In binary, the number of bits required for n is approximately log2(n).
    - Running time should be analyzed relative to the number of bits in the input, not just the value of n.
    - Plotting running time against log2(n) can reveal different growth trends compared to plotting against n.
    - For the Fibonacci problem, the output size (number of bits in Fibonacci(n)) can be much larger than the input size (number of bits in n).

- **Explanation**
  The instructor explains that for numerical problems, the size of the input should be measured by the number of bits needed to represent the number, not the number itself. For example, 32 in binary is 100000, which is 6 bits. The running time of an algorithm should be considered in terms of the input's bit-length. The instructor demonstrates plotting running time against both n and log2(n), showing that the trends can differ. For large n, the output (Fibonacci(n)) can require many more bits than the input.
- **Measuring Input Size for Fibonacci Computation**

  > Given n = 32, the number of bits required is log2(32) = 5. For n = 33, it's slightly more than 5, but 5 bits suffice. The instructor plots running time versus both n and log2(n) to compare trends.

  i. Recognize that input size for numerical problems is the number of bits, not the value.
  ii. Calculate the number of bits for various n values.
  iii. Plot running time against both n and log2(n).
  iv. Observe that the running time trend changes depending on the x-axis measurement.

6. Output Size Considerations in Algorithm Analysis

- **Key Points**
  - Given log2(n) bits as input, the output (Fibonacci(n)) may require significantly more bits.
  - For example, inputting 5 bits (n = 32) and computing Fibonacci(32) results in an output that requires many more bits.
  - This highlights the importance of considering both input and output sizes in algorithm analysis.
- **Explanation**
  The instructor poses the question: if you are given log2(n) bits representing n, and asked to compute Fibonacci(n), how many bits are needed to represent the output? For n = 32 (5 bits), Fibonacci(32) is a much larger number, requiring more bits to store. This demonstrates that output size can grow much faster than input size, which is important for understanding the limits of algorithm efficiency.
- **Output Size for Fibonacci(32)**

  > Given n = 32 (5 bits), compute Fibonacci(32). The output, Fibonacci(32), is a large number that requires more bits to represent than the input.

  i. Input: 5 bits (n = 32).
  ii. Compute Fibonacci(32).
  iii. Determine the number of bits required to represent Fibonacci(32).

iv. Conclude that output size can be much larger than input size.

7. Growth of Fibonacci Numbers and Bit Representation

- **Key Points**
  - Fibonacci(n) is greater than or equal to 2^(n/2) as n becomes larger.
  - To represent Fibonacci(n), at least n/2 bits are needed.
  - Input size (n) can be given in log2(n) bits, but the output (Fibonacci(n)) may require n/2 bits.
  - If b = log2(n), then n = 2^b, and n/2 = 2^(b-1).
  - Increasing the input by 1 bit doubles the amount of work required.
  - The time complexity for array-based Fibonacci algorithms is proportional to n, which is 2^b when input is b bits.
  - This leads to exponential growth in computation time as input size increases.
- **Explanation**
  The class discusses how the Fibonacci sequence grows and how this affects the number of bits needed to represent its values. When n increases, Fibonacci(n) grows faster than linearly, requiring more bits for representation. If the input is given in b bits (where b = log2(n)), the output may require 2^(b-1) bits. This means that for every additional bit in the input, the work required doubles. The array-based algorithm for Fibonacci takes time proportional to n, which translates to exponential time in terms of input bits.
- **Bit Growth Example with Fibonacci Numbers**

  > If you look at n = 22, the execution time is around 3.5 million nanoseconds. When n increases to 23, the execution time is about 7 million nanoseconds. At n = 24, the time almost doubles again. This demonstrates that each additional bit in the input doubles the work required.

  i. At n = 22, execution time is 3.5 million nanoseconds.
  ii. At n = 23, execution time is about 7 million nanoseconds.
  iii. At n = 24, execution time doubles again.
  iv. This pattern shows that each extra bit in the input leads to twice the computational work.

8. Measuring Algorithm Complexity by Input and Output Bits

- **Key Points**
  - Sorting an array of n integers: input size is 32n bits (assuming 32-bit integers), output size is also 32n bits.
  - For arrays, the number of bits per element is constant, so complexity is usually measured in terms of n.

    - For problems like Fibonacci, the output size in bits is not constant with respect to the input size.
    - The correct way to express time complexity is in terms of input bits: if n requires B bits, the algorithm may be O(2^B) or Θ(2^B).
    - It is incorrect to simply say an algorithm is O(n) without considering how n relates to the number of input bits.
- **Explanation**
  The class emphasizes that for most algorithms, such as sorting, the number of bits per element is constant, so the input and output sizes are proportional. However, for algorithms like those computing Fibonacci numbers, the output size in bits can be much larger than the input size. Therefore, algorithm complexity should be measured in terms of the number of bits in the input and output, not just the value of n. For example, if n is given in B bits, the algorithm may require O(2^B) time.
- **Sorting Array Example**
  > Given an array of n integers, each integer is 32 bits. The input size is 32n bits. Sorting the array produces another array of 32n bits. The input and output sizes are the same, so the focus is on the number of elements (n), not the number of bits per element.

  i. Input: n integers, each 32 bits, total 32n bits.
  ii. Output: sorted array, also 32n bits.
  iii. The number of bits per element is constant, so complexity is measured in terms of n.

9. Prime Number Checking Algorithm and Complexity

- **Key Points**
    - To check if n is prime, test divisibility from 2 up to n-1.
    - Assuming division is constant time, the algorithm takes up to n steps.
    - Input size is log2(n) bits, so n = 2^b if input is b bits.
    - The algorithm may require up to 2^b steps, which is exponential in the number of input bits.
    - If n is prime, the algorithm checks all numbers from 2 to n-1.
    - A more efficient approach is to check up to the square root of n.
- **Explanation**
  The class transitions from Fibonacci numbers to prime number checking. The naive algorithm checks divisibility from 2 to n-1, which takes n steps. Since the input is given in log2(n) bits, the number of steps is exponential in the input size. For large n, this becomes impractical. The class suggests that a more efficient algorithm would only check up to the square root of n.
- **Naive Prime Checking Algorithm**

> For i = 2 to n-1, check if n is divisible by i. If n is not divisible by any i, then n is prime. If n is prime, the algorithm performs n-2 checks.

i. Input: n (given in log2(n) bits).
ii. Loop from i = 2 to n-1.
iii. Check if n % i == 0.
iv. If no divisors found, n is prime.
v. Worst-case: n-2 checks, which is up to 2^b steps if input is b bits.

10. Prime Number Checking and Square Root Optimization

- **Key Points**
  - If n has a factor less than or equal to square root of n, it must have a corresponding factor greater than or equal to square root of n.
  - For non-perfect squares, factors are distributed equally on either side of the square root.
  - It is sufficient to check divisibility up to and including the square root of n.
  - For perfect squares like 9, the square root itself is a factor.
  - The number of operations required is proportional to the square root of n.
- **Explanation**
  The reasoning is that for any composite number n, if you find a factor less than or equal to square root of n, the corresponding cofactor will be greater than or equal to square root of n. Therefore, checking up to square root of n is enough. For perfect squares, the square root itself is a factor, so the check must include it. This optimization reduces the number of checks needed compared to checking all numbers up to n.
- **Checking if 9 is Prime**
  > For n = 9, the square root is 3. Factors to check are 2 and 3. 9 is divisible by 3, so it is not prime.

  i. Calculate square root of 9, which is 3.
  ii. Check divisibility by 2: 9 % 2 != 0.
  iii. Check divisibility by 3: 9 % 3 == 0.
  iv. Since 9 is divisible by 3, it is not prime.

11. Algorithmic Complexity Relative to Input Size in Bits

- **Key Points**
  - Square root of n is n to the power of 1/2, or 2 to the power of b/2 if n is represented with b bits.
  - The running time is still exponential in b, specifically 2 to the power of b/2.
  - For every extra two bits in the input, computation time doubles.

    - For every extra four bits, computation time quadruples.
    - This exponential growth is significant for cryptographic security.
- **Explanation**
  If n is represented with b bits, then n = 2^b. The square root of n is 2^(b/2). Thus, the number of checks required for primality is exponential in the number of bits. This means that increasing the key size by just a few bits can dramatically increase the time required to factor n, which is crucial for the security of cryptographic systems.
- **Doubling Computation Time with Bit Increase**
  > If you go from 1024 bits to 1026 bits, the computation time doubles. If you go from 1024 to 1028 bits, it quadruples.

  i. For 1024 bits, running time is proportional to 2^512.
  ii. For 1026 bits, running time is proportional to 2^513.
  iii. Each increase of two bits doubles the running time.

12. Importance of Prime Checking in Cryptography

- **Key Points**
    - Public key cryptography uses keys with two numbers: x and n.
    - n is the product of two primes, p and q.
    - Security relies on the difficulty of factoring n into p and q.
    - If an attacker can factor n, they can break the cryptosystem.
    - Key sizes commonly used: 256 bits (small), 1024 bits (common), 4096 bits (very secure).
- **Explanation**
  In systems like SSH and SSL, the public key includes a number n, which is the product of two large primes p and q. The security of these systems depends on the computational difficulty of factoring n to retrieve p and q. As the number of bits in n increases, the time required to factor n increases exponentially, making the system more secure.
- **Security Scaling with Key Size**
  > If breaking a 1024-bit key (2^512 operations) takes three months, adding two bits (1026 bits) makes it take six months; adding another two bits (1028 bits) makes it take one year. Doubling from 1024 to 2048 bits increases the time to thousands of years.

  i. 1024 bits: 2^512 operations, takes three months.
  ii. 1026 bits: 2^513 operations, takes six months.
  iii. 1028 bits: 2^514 operations, takes one year.
  iv. 2048 bits: 2^1024 operations, takes thousands of years.

13. Algorithm Output Size and Its Impact on Running Time

- **Key Points**
  - Prime checking: input is b bits, output is 1 bit (0 or 1).
  - Fibonacci calculation: input is b bits, output is proportional to 2^b bits.
  - For Fibonacci, the output size is exponentially larger than the input.
  - Running time cannot be less than the time required to write the output to memory.
  - For prime checking, there is hope to make algorithms faster since output is much smaller than input.
- **Explanation**
  For prime checking, the answer is a single bit, so theoretically, the algorithm could be made faster. For Fibonacci, the output grows exponentially with the input size, so the running time must be at least as large as the output size, i.e., at least 2^b. This sets a lower bound on the running time for such problems.
- **Fibonacci Output Size**
  > Given b bits as input, the nth Fibonacci number requires output of at least 2^b bits, so the running time cannot be less than 2^b.

  i. Input: b bits.
  ii. Output: Fibonacci number, which is at least 2^b bits.
  iii. Running time must be at least enough to write 2^b bits to memory.

14. Measuring Algorithm Complexity: Time and Space

- **Key Points**
  - Algorithm complexity is measured relative to the number of bits in the input.
  - For arrays, the number of bits per element is constant, so focus is on n (number of elements).
  - For problems like Fibonacci and prime numbers, the number of bits in the input matters.
  - Running time can never be less than the amount of memory required for the output.
  - In some cases, space complexity is as important as time complexity.
  - The size of the program itself (in bits) can also affect overall complexity.
- **Explanation**
  When analyzing algorithms, it's important to consider both the time required to process the input and the space required to store the output. For most algorithms, the program size is constant and negligible, but in some cases, the program can grow with the problem size. The running time must always be at least as large as the space needed for the output.
- **Counting Operations in an Array Algorithm**

> When checking for duplicates in an array, you can count the number of operations (initialization, comparisons, etc.) and relate this to the input size n.

i. Write the algorithm for checking duplicates.
ii. Count the number of initializations and comparisons.
iii. Sum the total number of operations as a function of n.

15. Approximating Algorithm Running Time

- **Key Points**
  - t(n) represents the exact running time for input size n.
  - Exact t(n) may be hard to determine.
  - Approximations are used to analyze algorithm performance as n increases.
  - Big O analysis often ignores constants and focuses on growth rate.
- **Explanation**
  Since the exact running time function t(n) can be complex or unknown, we use approximations to understand how the algorithm scales with input size. Big O notation helps by focusing on the dominant term and ignoring constants, which is useful for comparing algorithms as input size grows.
- **Approximating t(n) for an Algorithm**
  > Given a plot of n vs. t(n), the curve may be complex, so we approximate it with a simpler function to analyze growth.

  i. Plot n vs. t(n) for the algorithm.
  ii. Observe the shape of the curve.
  iii. Approximate with a function (e.g., linear, quadratic, exponential) for analysis.

16. Big O, Omega, and Theta Notations

- **Key Points**
  - Big O notation (O(f(n))) describes an upper bound for the running time of an algorithm. It includes all functions whose growth rate is less than or equal to f(n) after a certain point n0, with a constant multiplier a > 0.
  - Omega notation (Ω(g(n))) describes a lower bound for the running time. It includes all functions whose growth rate is at least g(n) after a certain point n1, with a constant multiplier b > 0.
  - Theta notation (Θ(f(n))) describes a tight bound, meaning the running time is bounded both above and below by f(n), with different constants a and c, for all n greater than some n0.
  - Big O, Omega, and Theta are sets of functions, not individual functions.
  - The goal is to find a tight (Theta) bound for the running time, but if not possible, Big O (upper bound) is used.

- Constants in these notations (like a, b, c) are ignored in asymptotic analysis, but large constants can make Big O less informative.
- The running time is measured with respect to the input size, often in terms of the number of bits.
- Upper and lower bounds should be as close as possible for the bound to be useful.
- If the upper and lower bounds are the same function (up to constant factors), then the running time is in Theta of that function.

- **Explanation**
  The class began by introducing the concept of bounding the running time of algorithms using functions. If a function f(n) multiplied by a constant a is strictly greater than another function p(n) for all n > n0, then p(n) is in the set O(f(n)). This means f(n) is an upper bound for p(n). Similarly, if another function g(n) multiplied by a constant b is always less than p(n) after some point n1, then p(n) is in the set Ω(g(n)), making g(n) a lower bound. The ideal scenario is when the upper and lower bounds are tight, i.e., the same function up to constant factors, which is denoted by Θ(f(n)). The instructor emphasized that these notations are sets of functions, not individual functions, and that the running time of an algorithm is one of the functions in these sets. The importance of tight bounds (Theta) was highlighted, as loose bounds (Big O or Omega alone) can be uninformative. Examples were given to illustrate the difference between useful and unhelpful bounds, such as comparing n! and 2^n, or giving salary bounds like less than $1 billion or more than $0. The instructor also discussed the significance of measuring input size in bits and the impact of large constants in asymptotic notation.
- **Example of Big O, Omega, and Theta Bounds**
  > Suppose there is a function f(n) and a constant a > 0 such that a * f(n) > p(n) for all n > n0. Then, p(n) is in O(f(n)). Similarly, if there is a function g(n) and a constant b > 0 such that b * g(n) < p(n) for all n > n1, then p(n) is in Ω(g(n)). If there exist constants a and c such that a * f(n) < t(n) < c * f(n) for all n > n0, then t(n) is in Θ(f(n)).

  i. Identify the function to be bounded (e.g., t(n)).
  ii. Find an upper bound function f(n) and a constant a such that a * f(n) > t(n) for all n > n0.
  iii. Find a lower bound function g(n) and a constant b such that b * g(n) < t(n) for all n > n1.
  iv. If both bounds can be established with the same function f(n) (up to different constants), then t(n) is tightly bounded (Theta).
- **Comparing n! and 2^n as Bounds**
  > The instructor asked: Is n! bigger or smaller than 2^n? n! is larger than 2^n because n! multiplies n numbers, most of which are greater than 2, while 2^n is n numbers, each of which is 2. If the running time is proportional to 2^n, it is

> technically O(n!), but this is not a useful bound because n! is much larger than 2^n.

i. Compare the growth rates of n! and 2^n.
ii. Recognize that using n! as an upper bound for a function that grows like 2^n is technically correct but not informative.
iii. Understand that a useful bound should be as tight as possible to the actual growth rate.

- **Salary Analogy for Bounds**

> The instructor gave an analogy: If asked about salary and the answer is 'less than $1 billion,' it is technically an upper bound but not useful. Similarly, saying 'more than $0' is a lower bound but also not useful.

i. Relate the concept of upper and lower bounds to real-world examples.
ii. Demonstrate that overly loose bounds provide little practical information.
iii. Emphasize the need for tight bounds in algorithm analysis.

Homework

[ ] Think about the stable matching algorithm implementation and be prepared to discuss its correctness and analysis on 2025-07-17.
[ ] Form groups of three for the programming assignment. A link will be sent out to indicate your teams. Details for the programming assignment will be provided after the situation. No class on Friday, July 18, 2025. If you have questions about today's material, ask the instructor on July 17, 2025, if not today.

> **AI Suggestion**
>
> - The core of this lesson is understanding Asymptotic Analysis and Algorithm Optimization. It's recommended to start with implementing and analyzing recursive and dynamic programming solutions for the Fibonacci sequence to grasp these concepts through hands-on coding and runtime comparison.
> - Core content of Asymptotic Analysis: Method for evaluating algorithm efficiency by analyzing runtime as input size increases, focusing on speed comparison.
> - Core content of Algorithm Optimization: Dynamic Programming for Fibonacci: Optimization using dynamic programming (bottom-up approach) reduces time complexity from exponential to linear by storing previously computed values.
> - Additionally, here are some extracurricular resources:
> - Practical application of Asymptotic Analysis and Dynamic Programming: https://leetcode.com/problems/fibonacci-number/
> - Alternative perspective on Asymptotic Analysis: https://visualgo.net/en/complexity