07-17 Education Class: Cryptography and Sorting Algorithms

> Date & Time: 2025-07-17 09:07:16
> Location: [Insert Location]
> Course Name: [Enter Course Name]

Keywords

`Cryptography` `Sorting Algorithms` `Algorithm Analysis`

Key Learnings

1. Introduction to Cryptography: Cryptography is the practice of securing communication to ensure that only authorized individuals can access a message. It addresses the problem of an eavesdropper intercepting and understanding a message sent between a sender and a receiver.
2. Symmetric Key Cryptography: A method of cryptography where the sender and receiver use the same secret key to both encrypt and decrypt a message. This is called symmetric key cryptography because the key is identical on both sides.
3. The Key Exchange Problem: The primary challenge of symmetric key cryptography is the secure distribution of the shared key. If the sender and receiver have never met, they cannot pre-share a key or a codebook, which makes establishing a secure communication channel difficult.
4. Efficient Exponentiation (Repeated Squaring): Calculating A to the power of B (A^B) can be performed much more efficiently than the naive method of multiplying A by itself B times. By using a method called repeated squaring, the computation time becomes proportional to the number of bits in B (log B), not the value of B itself.
5. RSA Public Key Cryptography Scheme: RSA is an asymmetric cryptography system where the encryption key is made public and the decryption key is kept private. Its security relies on the computational difficulty of factoring the product of two large prime numbers.
6. Public-Key Cryptography and RSA: An overview of public-key cryptography, where a public key (e, n) is used for encryption and a private key (d) is used for decryption. The security relies on the computational difficulty of factoring a large number n into its prime factors p and q.

7. Fermat’s Little Theorem: A fundamental theorem in number theory which states that if p is a prime number, then for any integer a not divisible by p, a raised to the power of p-1 is congruent to 1 modulo p. That is, $a^{(p-1)} \equiv 1 \pmod{p}$.
8. Linear Search Algorithm: A basic search algorithm that finds an item in a list or array by checking each element one by one, from the beginning, until the target element is found or the end of the list is reached.
9. Binary Search Algorithm: An efficient search algorithm that finds the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half.
10. Binary Search: Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you’ve narrowed the possible locations to just one.
11. Fundamentals of Sorting: Sorting is the process of arranging items in a sequence ordered by some criterion. We focus on comparison-based sorting, where elements are ordered by comparing them to each other.
12. Insertion Sort: Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It iterates through an input array and for each element, it finds its correct position in the already sorted part of the array and inserts it there.
13. Divide and Conquer Strategy: Divide and Conquer is an algorithmic paradigm where a problem is solved by breaking it down into smaller sub-problems, solving them recursively, and then combining their solutions to solve the original problem.
14. Merge Sort: Merge sort is an efficient, comparison-based, divide and conquer sorting algorithm. It divides the unsorted list into n sub-lists, each containing one element (a list of one element is considered sorted), and then repeatedly merges sub-lists to produce new sorted sub-lists until there is only one sub-list remaining.
15. Quick Sort: Quick sort is another efficient, comparison-based, divide and conquer sorting algorithm. It works by selecting a ‘pivot’ element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
16. Practical Performance of Sorting Algorithms: A practical comparison of sorting algorithms shows a significant performance difference between algorithms with n-squared complexity and those with n log n complexity, especially for large datasets.
17. Using LLMs for Coding Assistance: An overview of how to effectively use Large Language Models (LLMs) as a tool to assist with various coding tasks.
18. Logarithm Problem Warm-up: A quick warm-up problem involving logarithmic rules, noted as being similar to a problem from the previous day’s session on July 16, 2025.
19. Proving log n! is in Theta(n log n): This is a continuation of a problem from the previous day (July 16, 2025), where it was shown that log n! is in Big O of n log n. The current goal is to show the other direction of the proof to establish that log n! is in Theta(n log n).

Explanations

1. Introduction to Cryptography

- **Key Points**
    - The purpose of cryptography is to ensure secret communication between a sender and a receiver.
    - It solves the problem of an eavesdropper intercepting messages sent over a network.
    - The basic solution is to encrypt the message, making it unreadable to outsiders.
    - Simple ciphers, like letter substitution, are vulnerable to frequency analysis.
    - Encryption should be fast for the sender, and decryption should be fast for the receiver.
- **Explanation**
  The fundamental premise involves a sender (S) and a receiver ®. Any message can be converted into a sequence of bits. If sent directly, an eavesdropper can intercept and read it. To prevent this, the message must be encoded or encrypted. Simple ciphers, like substituting letters, can be broken by analyzing letter frequencies (e.g., ‘E’ is the most common letter in English). More complex systems were used historically, such as during World War II. The goal is to have a system where encryption and decryption are fast for the sender and receiver but difficult for others to break.

2. Symmetric Key Cryptography

- **Key Points**
    - The sender and receiver share a single secret key.
    - Encryption and decryption are inverse operations.
    - A common implementation uses the XOR operation.
    - The security of the system depends on the secrecy of the key.
    - If the same key is used repeatedly, it can become vulnerable to pattern analysis.
    - For maximum security, keys should be used only once, requiring a pre-shared codebook of keys.
- **Explanation**
  The sender and receiver must agree on a shared secret key, which is a sequence of bits. A simple and effective way to encrypt is to use the bitwise XOR (exclusive OR) operation. The sender XORs the message bits with the key bits to produce the encrypted message. The receiver, having the same key, performs the exact same XOR operation on the encrypted message to recover the original message. The security of this system relies entirely on keeping the shared key secret. If the same

key is used for many messages, patterns can emerge, allowing an attacker to potentially figure out the key. For very high security, a key should be used only once, which requires the sender and receiver to have a long ‘codebook’ of keys.

- **XOR Encryption Example**

  > Message (M): 110110
  > Key (K): 011011
  > Encrypted Message (M XOR K): 101101
  > Decryption (Encrypted Message XOR K): 101101 XOR 011011 = 110110 (recovers the original message)

  i. The XOR (exclusive OR) operation is used for both encryption and decryption.
  ii. XOR logic: If two bits are the same (0 and 0, or 1 and 1), the result is 0. If they are different (0 and 1), the result is 1.
  iii. The sender encrypts the message by performing an XOR operation between the message and the key.
  iv. The receiver decrypts the message by performing the same XOR operation between the encrypted message and the same key.

3. The Key Exchange Problem

- **Key Points**
  - Symmetric key cryptography requires a pre-shared secret key.
  - This is difficult if the communicating parties have never met in person.
  - Managing separate codebooks for hundreds of contacts is impractical.
  - Using a third party to distribute keys introduces trust and security issues.
  - The solution is public key (asymmetric) cryptography, where the encryption and decryption keys are different.
- **Explanation**
  For symmetric cryptography to work, the sender and receiver must have a pre-arranged shared key. This could be done by meeting in person and exchanging a codebook. However, this is impractical for communicating with many people or with someone you’ve never met, such as in online transactions. Using a trusted third party to distribute keys introduces a new vulnerability, as the third party could be compromised or untrustworthy. This problem is addressed by public key cryptography, which is an asymmetric system where the sender and receiver use different keys.

4. Efficient Exponentiation (Repeated Squaring)

- **Key Points**
  - Naive exponentiation (A^B) takes theta(B) time, which is exponential in the number of bits of B.

    - A more efficient method is repeated squaring.
    - This method reduces the time complexity to theta(log B), which is linear in the number of bits of B.
    - This efficiency is critical for public key cryptography systems like RSA.
- **Explanation**
  The naive algorithm for A^B involves a loop that runs B times, making it an exponential time algorithm in terms of the number of input bits for B. A faster method, known as repeated squaring, involves representing the exponent B in binary. For example, A^12 is A^(8+4). This can be computed by first calculating A^2, then A^4 (by squaring A^2), then A^8 (by squaring A^4), and finally multiplying A^8 and A^4. This process involves a number of steps proportional to the number of bits in the exponent B (which is log B), making exponentiation a relatively fast operation, which is crucial for algorithms like RSA.
- **Example of computing A to the power of 12**
  > The exponent 12 in binary is 1100. This corresponds to 8 + 4. Therefore, A^12 = A^8 * A^4.

  i. a. Start with A.
  ii. b. Square it to get A^2.
  iii. c. Square A^2 to get A^4.
  iv. d. Square A^4 to get A^8.
  v. e. Multiply the components corresponding to the ‘1’ bits in the binary representation of the exponent: A^4 * A^8.
  vi. This requires 3 squarings and 1 multiplication, which is far fewer than the 11 multiplications required by the naive method.

5. RSA Public Key Cryptography Scheme

- **Key Points**
    - Find two large prime numbers, p and q.
    - Compute n = p * q.
    - Find an integer ‘e’ such that the greatest common divisor of e and (p-1)*(q-1) is 1. ‘e’ is part of the public key.
    - Compute ‘d’ such that d is the modular inverse of e modulo (p-1)*(q-1). ‘d’ is the private key.
    - The use of keys larger than 768 bits is restricted in some regions, like the US, and is treated with the same import regulations as missiles.
- **Explanation**
  The setup for RSA involves several steps. First, find two very large prime numbers, p and q (e.g., 768, 1024, or 2048 bits long). Compute their product, n = p * q. Then, find a number ‘e’ (for encryption) such that it is co-prime with (p-1)*(q-1); this means*

their greatest common divisor is 1. Finally, compute the decryption key ‘d’ as the modular multiplicative inverse of ‘e’ modulo (p-1)(q-1). The public key consists of (n, e) and the private key is d. The security of the system comes from the fact that while exponentiation (used for encryption and decryption) is fast, finding p and q given only n (factoring) is extremely difficult.

- **Modular Inverse Concept Example**

  > Suppose (p-1)*(q-1) is 32 and e is 3. We need to find a number d such that (d * e) mod 32 = 1. In this case, we are looking for d where (d * 3) mod 32 = 1.

  i. This example illustrates the concept of finding the modular multiplicative inverse.
  ii. This step is essential for generating the private key ‘d’ from the public exponent ‘e’ and the product of (p-1) and (q-1).
  iii. In real numbers, the inverse of 5 is 1/5. In modular arithmetic with integers, the inverse ‘d’ of a number ‘e’ is an integer such that (d * e) gives a remainder of 1 when divided by the modulus.

6. Public-Key Cryptography and RSA

- **Key Points**
  - The public key (e, n) is for encryption and is shared publicly.
  - The private key (d, p, q) is for decryption and is kept secret.
  - The security of RSA relies on the difficulty of factoring the large number n into its prime factors p and q.
  - Prime numbers p and q are very large, such as 512 or 1024 bits.
  - The private key ‘d’ is the inverse of ‘e’ modulo (p-1)(q-1).
  - Long messages are broken into smaller chunks, like 8,000 bits, to fit within the modulo n.
  - Fast exponentiation algorithms are used to make encryption and decryption efficient.
  - Practical applications include SSH keygen, SSL/TLS for HTTPS, and digital signatures.
  - For performance, public-key crypto is often used to securely exchange a symmetric key, which is then used for bulk encryption with a faster algorithm like XOR.
- **Explanation**
  The process starts by choosing two large prime numbers, p and q. The modulus n is calculated as n = p * q. An integer e is chosen to be co-prime with (p-1)(q-1). The private key d is then calculated as the modular multiplicative inverse of e modulo (p-1)(q-1), meaning e*d ≡ 1 (mod (p-1)(q-1)). The public key consists of (e, n), which is made public. The private key consists of (d, p, q), which is kept secret. To encrypt a message M, the sender computes C = M^e mod n. To decrypt, the receiver

computes M = C^d mod n. This works because C^d = (M^e)^d = M^(ed). Since ed = 1 + k(p-1)(q-1), this simplifies to M modulo n, based on Fermat's Little Theorem. After key generation, p and q can be discarded, and only d and n are needed for the private key.

- **Example of finding the private key 'd'**

  > Suppose e = 3 and (p-1)(q-1) = 32.

  i. We need to find d such that e*d ≡ 1 (mod 32).
  ii. In this case, d = 11, because 3 * 11 = 33, and 33 modulo 32 is 1.
  iii. So, 11 is the inverse of 3 modulo 32.
- **Digital Signature Example**

  > To sign a document M, the signer computes a signature S = M^d mod n using their private key.

  i. The signer sends the document M and the signature S.
  ii. A verifier with the public key (e, n) can then compute MDE modulo N on the signature and verify that it matches the original document M.
  iii. If they are the same, they know it could have only been encrypted with the sender's private key, and therefore trust that they actually sent the message.
- **SSL Certificate Expiration**

  > When using HTTPS, a browser might show a 'certificate has expired' message.

  i. A certificate is essentially a website's public key, registered with a third-party provider like VeriSign for a fee (e.g., $100 per year).
  ii. This third party attests that the public key belongs to the website.
  iii. If the payment is late or the registration period ends, the certificate expires, and browsers will warn users.

7. Fermat's Little Theorem

- **Key Points**
  - Statement: a^(p-1) ≡ 1 (mod p) for a prime p and a not a multiple of p.
  - It is the key mathematical result that makes RSA decryption possible.
  - It is distinct from Fermat's Last Theorem.
- **Explanation**
  The theorem is a crucial result that underpins why RSA decryption works. The decryption calculation M^(ed) mod n relies on the property that ed ≡ 1 (mod (p-1)(q-1)). This can be written as ed = 1 + k(p-1)(q-1). When we compute M^(ed), we are computing M^(1 + k(p-1)(q-1)). Using properties derived from Fermat's Little Theorem, such as M^(p-1) ≡ 1 (mod p) and M^(q-1) ≡ 1 (mod q), it can be shown that M^(k(p-1)(q-1)) ≡ 1 (mod pq). Therefore, M^(ed) ≡ M^1 * 1 ≡ M (mod n), which recovers the original message. The theorem can be proven by induction on 'a'.

- **Numerical Example of Fermat’s Little Theorem**

  > Let’s test with a = 4 and p = 3. We calculate a^(p-1) mod p, which is 4^(3-1) mod 3.

  i. 4^2 is 16.
  ii. 16 modulo 3 is 1, which confirms the theorem.

8. Linear Search Algorithm

- **Key Points**
  - It works on any list, regardless of whether it is sorted or not.
  - The time complexity in the worst-case scenario is proportional to the length of the array.
  - The complexity is described as O(n) or, more precisely, Theta(n) in the worst case, where n is the number of elements in the array.
- **Explanation**
  To perform a linear search, you iterate through the array using a loop, from the first index (0) to the last. In each iteration, you compare the current element with the target value. If they match, the search is successful, and you can return true or the index of the element. If the loop completes without finding the element, the search is unsuccessful.
- **Code Logic for Linear Search**

  > for i equals 0, i less than a dot length, or something like that, i plus plus, you’re going to check if, I mean, if this is returning a visit in the array, sometimes I might return the location in the array, things like that, if it’s not, obviously, okay.

  i. This loop iterates through each element of the array.
  ii. It compares each element to the target value.
  iii. If a match is found, the function stops and indicates success.
  iv. If the loop finishes, it means the value was not in the array.

9. Binary Search Algorithm

- **Key Points**
  - The array must be sorted for binary search to work.
  - It is significantly faster than linear search for large arrays.
  - The time complexity is logarithmic, O(log n), because the size of the problem is divided by two at each step.
  - The number of steps is related to the number of bits in the binary representation of the array’s length.
- **Explanation**
  Binary search requires the array to be sorted first. The process begins by comparing

the target value to the middle element of the array. If the target value matches the middle element, its position is found. If the target value is less than the middle element, the search continues on the lower (left) half of the array. If the target value is greater than the middle element, the search continues on the upper (right) half. This process is repeated, with the search interval being halved in each step, until the value is found or the interval is empty.

- **Example of Binary Search**

  > Search for the number 3 in the sorted array [1, 3, 5, 6, 7, 9].

  i. First, find the midpoint of the array. The array has six elements, so the midpoint is between index 2 (value 5) and 3 (value 6). Let's pick the element at index 2.
  ii. Compare the target value (3) with the middle element's value (5).
  iii. Since 3 is smaller than 5, the search is narrowed to the left half of the array, which contains all the smaller values: [1, 3].
  iv. Repeat the process on the new, smaller array until the element is found.

10. Binary Search

- **Key Points**
  - Binary search operates on sorted arrays.
  - It repeatedly divides the search interval in half.
  - The time complexity in the worst case is theta(log n to the base 2).
  - It offers a huge performance improvement over linear search (O(n)). For an array of 1024 elements, binary search takes about 10 comparisons, whereas linear search could take up to 1024.
  - The prerequisite for binary search is that the array must be sorted.
- **Explanation**
  The core idea of binary search is to reduce the problem size by half in each step. For an array of N elements, the number of times you can divide it by 2 is approximately log N. This is why the algorithm's time complexity is logarithmic. Integer division by 2 is computationally equivalent to a right bit shift, which is a very fast operation. Modern compilers often optimize integer division by a constant 2 into a right shift instruction for better performance.
- **Integer Division by 2 as a Right Shift**

  > Let's say I have the number 5, which in binary is 101. I want to divide this by 2 using integer division. This is the same as a right shift. So, I throw away the last bit and I get 10 in base 2, which is 2. The actual result is 2.5, but in integer division, the fractional part is truncated.

  i. The number 5 in binary is 101.
  ii. Dividing by 2 is equivalent to a right bit shift.
  iii. Shifting 101 to the right by one bit results in 10.

iv. The binary number 10 is equal to the decimal number 2.
v. This demonstrates how division by 2 can be implemented as a fast bitwise operation.

11. Fundamentals of Sorting

- **Key Points**
  - Sorting helps in efficient searching and finding elements like the k-th largest.
  - Comparison-based sorting algorithms rely on comparing pairs of elements.
  - In-place sort: Uses a constant amount of extra memory, modifying the array directly.
  - Stable sort: Does not change the relative order of equal elements.
  - Lower Bound: The best possible worst-case time complexity for comparison-based sorting is n log n.
- **Explanation**
  Sorting is a fundamental operation in computer science, often used to enable other algorithms, like binary search. There are key properties to define sorting algorithms. An ‘in-place’ sort modifies the input array directly without requiring substantial extra memory. A ‘stable’ sort maintains the original relative order of elements that are considered equal. The theoretical lower bound for any comparison-based sorting algorithm is omega(n log n), meaning no such algorithm can be faster than n log n in the worst case.
- **Stable vs. Unstable Sorting**

  > Suppose you have an array of key-value pairs: (8, ‘fox’), (8, ‘cow’). We are sorting based on the number. In the original array, ‘fox’ comes before ‘cow’. A stable sorting algorithm will produce an output where (8, ‘fox’) still appears before (8, ‘cow’). An unstable algorithm might swap their order to (8, ‘cow’), (8, ‘fox’).

  i. The elements are considered equal because we are only comparing the numerical part (8).
  ii. A stable sort preserves the initial order of these equal elements.
  iii. An unstable sort does not guarantee the preservation of this order.
  iv. Stability is useful when sorting data on multiple criteria. For example, if you first sort a list of students by age, and then perform a stable sort by name, students with the same name will remain ordered by their age.

12. Insertion Sort

- **Key Points**
  - Best-case running time: Theta(n), when the array is already sorted.

    - Worst-case running time: Theta(n squared), when the array is sorted in reverse order.
    - Average-case running time: Theta(n squared).
    - It is an in-place sorting algorithm.
    - It can be implemented as a stable sort.
- **Explanation**
  The algorithm maintains a sorted sub-array at the beginning of the list. It takes the next element from the unsorted part and 'inserts' it into the sorted sub-array by shifting all the larger elements to the right. This process is repeated for all elements. For each of the n elements, it may have to search for the correct position (which can take up to n steps) and then shift other elements (another n steps in the worst case), leading to an n-squared complexity.
- **Insertion Sort Walkthrough**
  > Consider the array [2, 5, 4, 3, 8, 7, 1, 6].

1. Start with [2] as the sorted part.
2. Take 5. It's in the correct place. Sorted part: [2, 5].
3. Take 4. Insert it before 5. Sorted part: [2, 4, 5].
4. Take 3. Insert it between 2 and 4. Sorted part: [2, 3, 4, 5].
5. Continue this process for 8, 7, 1, and 6 until the entire array is sorted.
6. The algorithm iterates from the second element to the end of the array.
7. At each step 'k', the element at index 'k' is placed in its correct sorted position within the sub-array from index 0 to k.
8. This involves finding the correct insertion point and shifting elements to make space.

13. Divide and Conquer Strategy

- **Key Points**
    - It's a powerful technique for designing efficient algorithms.
    - Merge Sort splits the array and combines results by merging.
    - Quick Sort partitions the array and solves the sub-problems on the partitions.
- **Explanation**
  The general strategy for Divide and Conquer is:

1. **Divide:**Break the given problem into sub-problems of the same type.
2. **Conquer:**Recursively solve these sub-problems. If a sub-problem is small enough, solve it directly.
3. **Combine:**Combine the solutions of the sub-problems to get the solution for the original problem.
   This strategy is used in efficient algorithms like Merge Sort and Quick Sort.

14. Merge Sort

- **Key Points**
  - It is a divide and conquer algorithm.
  - Time complexity is theta(n log n) in all cases (best, average, worst).
  - It can be implemented as a stable sort.
  - It is not an in-place sort as it typically requires extra space to store the merged sub-arrays.
- **Explanation**
  The algorithm first recursively splits the array into two halves until it gets down to arrays of size one. An array with one element is inherently sorted. Then, it starts the 'merge' phase. The merge process combines two sorted sub-arrays into a single sorted array. This is done by comparing the first elements of each sub-array and picking the smaller one to be the next element in the final merged array. This continues until all elements are merged. The workhorse of the algorithm is the merge step, which takes theta(n) time to merge n elements. Since the array is split in half each time, there are log n levels of recursion. With theta(n) work at each level, the total time complexity is theta(n log n).
- **Merge Operation**
  > To merge two sorted arrays, say [1, 3, 5, 7, 8] and [2, 4, 6] :

1. Compare 1 and 2. Pick 1. Array 1 pointer moves to 3.
2. Compare 3 and 2. Pick 2. Array 2 pointer moves to 4.
3. Compare 3 and 4. Pick 3. Array 1 pointer moves to 5.
4. Compare 5 and 4. Pick 4. Array 2 pointer moves to 6.
5. Compare 5 and 6. Pick 5. Array 1 pointer moves to 7.
6. Compare 7 and 6. Pick 6. Array 2 pointer is now empty.
7. Copy the remaining elements from Array 1 (7, 8).
   Final merged array: [1, 2, 3, 4, 5, 6, 7, 8].
8. The merge operation takes two sorted arrays and combines them into a single sorted array.
9. It uses pointers (indices) for each array, starting at the beginning.
10. It compares the elements at the current pointers and copies the smaller element to the output array, then advances the corresponding pointer.
11. This process is repeated until one of the arrays is exhausted.
12. Finally, it copies the remaining elements from the non-exhausted array.
13. Merging a total of n elements takes theta(n) time.

15. Quick Sort

- **Key Points**

    - It is a divide and conquer algorithm.
    - Its performance depends heavily on the pivot selection strategy.
    - Average-case time complexity is theta(n log n).
    - Worst-case time complexity is theta(n squared), though this is rare with good pivot selection.
    - It is typically an in-place sort.
- **Explanation**
  Unlike Merge Sort which divides the array in half, Quick Sort's division depends on the chosen pivot. After picking a pivot, the array is rearranged so that all elements smaller than the pivot are placed before it, and all elements greater are placed after it. This is the 'partition' step. After partitioning, the pivot is in its final sorted position. The algorithm then recursively applies the same process to the sub-arrays of elements with smaller and larger values.

16. Practical Performance of Sorting Algorithms

- **Key Points**
    - There is a huge performance difference between n-squared and n log n algorithms for large inputs.
    - Bubble Sort and Insertion Sort are impractical for large datasets.
    - Merge Sort, Quick Sort, and standard library sorting functions are much more efficient.
    - The need for efficient algorithms became critical around the year 2000 with the rise of large-scale data processing in search engines and databases.
- **Explanation**
  An experiment was conducted comparing Bubble Sort, Insertion Sort (both n-squared) with Merge Sort, Quick Sort, and Java's built-in sort (all n log n). For small arrays, the difference is negligible. However, as the array size increases (e.g., to 65,000 or a million elements), the n-squared algorithms become extremely slow, while the n log n algorithms remain efficient. The graphs show that the running time for n-squared algorithms spikes dramatically, whereas for n log n algorithms, it grows much more slowly. This highlights the practical importance of choosing efficient algorithms for problems involving large inputs.
- **Sorting Algorithm Performance Experiment**
  > The experiment compared Bubble, Insertion, Merge, Quick, and Java's default sort on array sizes from 16 up to about one million.

- For n-squared algorithms (Bubble, Insertion), the test was stopped at 65,000 elements because they became too slow.
- The graph of running times shows a massive spike for Bubble and Insertion sort as the array size grows.

- A second graph, focusing on the faster n log n algorithms (Merge, Quick, Java’s sort) on a logarithmic x-axis, shows that Merge Sort is slightly slower than Quick Sort and Java’s sort. Quick Sort was slightly faster than Java’s sort in this specific test with integers, but the difference was very small.
    i. The huge performance gap between n-squared and n log n algorithms is evident.
    ii. Going from 100,000 to 1,000,000 elements (a 10x increase in size), the running time for an n log n algorithm increases by a factor of roughly 10 * log(10), which is about 10 * 3.2 = 32. In contrast, an n-squared algorithm’s time would increase by a factor of 10*10 = 100.
    iii. This demonstrates why algorithmic complexity matters for scalability.

17. Using LLMs for Coding Assistance

- **Key Points**

    - LLMs can be used to search and understand large quantities of documentation for complex libraries, like K-Unit 5.
    - They can provide starter code for libraries you are unfamiliar with, such as JNN, which is helpful when you’re struggling to begin.
    - You can ask LLMs to explain data types or concepts, for example, by asking for a quick summary of how a map works in a specific language.
    - For effective queries, you should specify the programming language (e.g., Java), what you want in return (e.g., a quick summary or code), and the specific topic.
    - It is recommended to start a new instance for each new question to prevent artifacts from previous queries from influencing the new output.
    - LLMs are particularly effective at generating small functions or code snippets. For larger blocks of code, they may struggle and introduce errors. The recommended approach is to generate small snippets and then combine them manually.

- **Explanation**

- **Code Generation for a Prime Number Checker**

    > An example of using an LLM for code generation. The request is: ‘In Java, create a class called demo that has a main function to check if a given number is a prime number.’

    i. The LLM generates the requested Java code.
    ii. It may make minor adjustments based on conventions, such as capitalizing the class name to ‘Demo’.

iii. Testing the generated code is crucial. For example, when tested with the number 29, the code correctly identifies it as a prime number.
iv. However, when tested with 27, the LLM’s code incorrectly identifies it as a prime number, highlighting that generated code can contain subtle errors and must be verified.
v. This demonstrates that while LLMs are useful for generating small functions, you must carefully review and test the output for correctness.

18. Logarithm Problem Warm-up

- **Key Points**
- **Explanation**
  The core task is to apply logarithmic rules to simplify an expression involving the numbers 12 and 20. The strategy is to separate the terms, use the power rule to bring down an exponent from the term with 20 to get a constant, and then simplify the remaining log base 2 of 12. After isolating the constant, the remaining expressions will be equal.

19. Proving log n! is in Theta(n log n)

- **Key Points**
  - The starting point is the summation format for log n!: Σ log i from i=1 to n.
  - The sum is split into two smaller sums.
  - We analyze the terms from i = n/2 to n.
  - All terms in this range are greater than or equal to log(n/2).
  - There are n/2 terms in this range.
  - This leads to the conclusion that log n! is bounded below by a function proportional to n log n.
- **Explanation**
  To prove that n log n is a lower bound for log n!, we start by expressing log n! as a summation: log n! = Σ(log i) for i from 1 to n. We then split this sum into two parts. We analyze the second part of the sum, which includes terms where i goes from n/2 to n. In this range, every term log(i) is greater than or equal to log(n/2). Since there are n/2 terms in this part of the sum, the total value of this part is greater than or equal to (n/2) * log(n/2). This result shows that log n! is in Ω(n log n). Because we have already established that log n! is in O(n log n), we can conclude that log n! is in Θ(n log n).

> **AI Suggestion**
>
> - The core of this lesson is applying the **Divide and Conquer Strategy**. It’s recommended to start with a hands-on exercise to grasp this concept through

practice: physically write out the steps of splitting and merging a list of 10 numbers using the Merge Sort method on paper.

- Core content of **Divide and Conquer Strategy**: An algorithmic approach of breaking a problem into smaller sub-problems, solving them, and combining the results, used by Merge Sort and Quick Sort.
- Additionally, here are some extracurricular resources:
    - **Practical application of Sorting Algorithms**: https://visualgo.net/en/sorting
    - **Alternative perspective on Sorting Performance**: https://www.youtube.com/watch?v=es2T6KY45cA