**Lesson 1.**

**Intro What is meant by a problem belongs to the class P, NP, EXP or EXP-Complete? Can you give an example?**

P:problem that has polynomial time solution

Example: Sorting a list of numbers using algorithms like merge sort or quicksort is in P because they have polynomial time complexity.

NP:problem can be verify

Example: The Boolean satisfiability problem (SAT) is in NP. Given a Boolean formula, it's easy to verify if a proposed assignment of values to variables makes the formula true or false in polynomial time.

EXP:can’t have polynomial time solution

Example: Factoring large integers into their prime factors is believed to be in EXP. It's considered difficult because the best-known algorithms have exponential time complexity.

EXP-Complete: EXP-Complete (Exponential Time-Complete):A problem is EXP-Complete if it is at least as hard as the hardest problems in EXP. In other words, it is as hard as any problem in EXP in terms of time complexity.Example: The "Tiling Problem" is an example of an EXP-Complete problem. It involves covering a grid with tiles in a certain way, and it is believed to be as hard as any problem in EXP.

**What is meant by a problem has a feasible solution? Can you give an example for a problem that has feasible solution? Can you give an example for a problem that has no feasible solution?**

When we say a problem has a "feasible solution," it means there exists an algorithm or method that can efficiently solve the problem within a reasonable amount of time. A problem with no feasible solution means that there is no practical method to solve the problem within a reasonable timeframe.Examples:Problem with a feasible solution: Sorting an array.Problem with no feasible solution: n\*n chess

**Can you “prove” (give logical argument) Algorithm gcd will halt?**

Let's find the GCD of 48 and 18.

Start with the two integers: a = 48 and b = 18.

Calculate the remainder of a divided by b: 48 % 18 = 12.

Replace a with b and b with the remainder: a = 18, b = 12.

Calculate the remainder of a divided by b: 18 % 12 = 6.

Replace a with b and b with the remainder: a = 12, b = 6.

Calculate the remainder of a divided by b: 12 % 6 = 0.

At this point, b becomes 0. The algorithm explicitly terminates when b is zero.

Since b is now 0, the GCD is the current value of a, which is 6.

So, the GCD of 48 and 18 is 6, which is the result obtained using Euclid's algorithm.

**Do you think one day someone will be able to write a program to solve nxn Chess in polynomial time?**

No, because the problem belongs to the EXP problem.

**What is halting Problem? Why that is important in our discussion?**

The Halting Problem is a fundamental question in computer science that asks whether a program can determine if another program will eventually stop running or continue running forever. It's important in our discussion because it shows that there are limits to what can be determined algorithmically, highlighting the concept of undecidability in computation.

**Lesson 2.**

**Intro to Complexity Can you write a recursive algorithm (not Java Program) for**

**(a) Binary Search**

**(b) Linear Search**

**(c) Merge Sort**

**(d) Selection Sort**

**(e) Quicksort**

**(f) Quickselect**

**For all those algorithms can you write the recurrence relations递推关系? Can you solve it using Master Theorem?**

Case 1:• Recurrence form: T(n) = aT(n/b) + f(n), where a < b^c for some positive constant c.• Example: The recurrence relation for Merge Sort, T(n) = 2T(n/2) + O(n), falls under Case 1 because a (2) is less than b^c (2^1).Case 2:• Recurrence form: T(n) = aT(n/b) + f(n), where a = b^c for some positive constant c.• Example: The recurrence relation for QuickSort, T(n) = 2T(n/2) + O(n), fits Case 2 because a (2) equals b^c (2^1).Case 3:• Recurrence form: T(n) = aT(n/b) + f(n), where a > b^c for some positive constant c, and f(n) is asymptotically smaller.• Example: The recurrence relation for Strassen's Matrix Multiplication algorithm, T(n) = 7T(n/2) + O(n^2), falls into Case 3 because a (7) is greater than b^c (2^1), and f(n) (O(n^2)) is asymptotically smaller.

(a) Binary Search:Algorithm: Divide and conquer by comparing the middle element.Recurrence Relation: T(n) = T(n/2) + O(1)Master Theorem: Yes, applicable (Case 2).(b) Linear Search:Algorithm: Iterate through elements one by one.Recurrence Relation: T(n) = T(n-1) + O(1)Master Theorem: Not applicable.(c) Merge Sort:Algorithm: Divide and conquer with merging.Recurrence Relation: T(n) = 2T(n/2) + O(n)Master Theorem: Yes, applicable (Case 2).(d) Selection Sort:Algorithm: Find minimum iteratively and swap.Recurrence Relation: T(n) = T(n-1) + O(n)Master Theorem: Not applicable.(e) Quicksort:Algorithm: Divide and conquer with pivot selection.Recurrence Relation: T(n) = 2T(n/2) + O(n)Master Theorem: Yes, applicable (Case 2).(f) Quickselect:Algorithm: Quickselect to find the k-th smallest element.Recurrence Relation: T(n) = T(n/2) + O(n)Master Theorem: Yes, applicable (Case 2).

**What is the significance of a, b and k in the Master Theorem?**

In the Master Theorem:'a' represents the number of subproblems created by the algorithm.'b' represents how the input size is reduced when dividing the problem into subproblems.'k' signifies the exponent in the running time of the "combine" step.

**Can you visualize in your mind, binary search as an algorithm which has “width 1 and depth log n?” and thus has the time complexity O(log n)?**

**Can you visualize in your mind, linear search as an algorithm which has “width n and depth 1?” and thus has the time complexity O(n)?**

**Can you visualize in your mind, merge sort as an algorithm which has “width n and depth log n?” and thus has the time complexity O(nlog n)?**

**Can you visualize in your mind, selection sort as an algorithm which has “width n and depth n?” and thus has the time complexity O(n2 )?**

**Remember Toy Sort? What is its complexity? How O(nlog k) is in agreement with Quicksort’s time complexity?**

**Remember to visualize the “area” covered by the algorithm. Now you know why Quickselect is Θ(n)**

**Lesson 3. Probability**

**You have thrown two dices. Let X be the value of the “first dice” and Y be the value of second dice.**

**(a) What is the expected value of X?**

E(X) = (1 + 2 + 3 + 4 + 5 + 6) / 6 = 3.5

**(b) What is the expected value of X+Y?**

E[Y] = 3.5

E[X + Y] = E[X] + E[Y] E[X + Y] = 3.5 + 3.5 E[X + Y] = 7

**(c) What is the expected value of 3X + 4Y?**

E(3X + 4Y) = 3E(X) + 4E(Y) = 3 \* 3.5 + 4 \* 3.5 = 10.5 + 14 = 24.5

**(d) Is E(aX + bY) = aE(X) + bE(Y)?**

**What is meant by two events are independent?**

E(aX + bY) = aE(X) + bE(Y)

E(3X + 4Y) = 3E(X) + 4E(Y)

24.5 = 3 \* 3.5 + 4 \* 3.5

24.5 = 10.5 + 14

24.5 = 24.5

So, the equation E(aX + bY) = aE(X) + bE(Y) is satisfied.

**I have thrown a dice 5 times and I already saw faces 1, 2, 3, 4 and 5 exactly once. What is the probability that I will see face 6 in my sixth throw?**

The probability of rolling a fair six-sided die and getting each of the faces 1, 2, 3, 4, and 5 exactly once in the first five throws is 1/6^5, as there is a 1/6 chance of each of those outcomes happening in each of the five throws.

Now, for the sixth throw, you want to find the probability of rolling a 6. Since the die is fair, the probability of rolling a 6 in any single throw is 1/6.

So, the probability of getting the face 6 on the sixth throw, given that you've seen faces 1, 2, 3, 4, and 5 exactly once in the first five throws, is also 1/6.

Therefore, the probability of seeing face 6 in your sixth throw is 1/6.

**Remember the for formula 1 𝑝 and two more! Now how to prove all three of them.**

**???**

Probability of an Event Occurring (P(E)): Probability that event E happens. It's a number between 0 and 1, inclusive.

Probability of the Complement of an Event (P(E')): Probability that event E does not happen. P(E') = 1 - P(E).

Addition Rule for Mutually Exclusive Events: If two events E1 and E2 cannot happen at the same time, the probability of either happening is the sum of their individual probabilities: P(E1 or E2) = P(E1) + P(E2).

**Lesson 4: Average case analysis and Amortized cost analysis**

**What is inversion?**

Out of order

An "inversion" occurs when two elements in a sequence are in the reverse order compared to their natural or sorted order. For example, (5, 2) is an inversion in [5, 2, 6, 1, 3, 4].

**If there are n elements, what is the maximum number of inversions possible? Answer: n(n-1)/2. Can you prove?**

Yes, the maximum number of inversions in a sequence of n elements is n(n-1)/2.

To prove it briefly: The maximum number of inversions occurs when the sequence is in the completely reversed order. Starting with the first element, there are (n-1) elements that are greater than it, giving (n-1) inversions. Moving to the second element, there are (n-2) elements greater, adding (n-2) inversions, and so on. The total number of inversions is the sum of these values, which is (n-1) + (n-2) + ... + 1 = n(n-1)/2.

**What is the average number of inversions possible? Answer: n(n-1)/4. Can you prove?**

consider that when you have n elements in a random order, on average, each element will be involved in about (n-1)/2 inversions. Since there are n elements in total, the average number of inversions is n \* (n-1)/2. To get the average per pair, you divide this by 2, giving you n(n-1)/4.

**What is the average time complexity of any inversion bound algorithm? Why?**

O(n log n)

**What is a lower bound?**

A lower bound is the minimum limit or the best possible performance that cannot be surpassed for a given problem or algorithm. It represents the lower limit of efficiency for solving a particular problem.

**What is the lower bound of search (an unsorted array)?**

The lower bound of searching in an unsorted array is O(n), meaning that, in the worst case, you must examine all n elements to find a specific value in the array.

**What is the lower bound of inversion bound sorting algorithms?**

Ω(n log n).

**Can you name three inversion bound sorting algorithms and their best, average and worst case time complexities?**

Merge Sort:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Counting Sort:

Best Case: O(n + k), where k is the range of input values

Average Case: O(n + k)

Worst Case: O(n + k)

Bitonic Sort:

Best Case: O(n log² n)

Average Case: O(n log² n)

Worst Case: O(n log² n)

**What is the worst case time complexity of any inversion bound algorithm? Why?**

The worst-case time complexity of any inversion-bound algorithm is typically Θ(n²), where n is the number of elements in the input.

This is because inversion-bound algorithms, by definition, involve comparing pairs of elements and counting inversions, which can result in a quadratic time complexity in the worst case. In the worst-case scenario, the input data may be in reverse order, creating the maximum number of inversions, and the algorithm has to examine and compare each pair of elements. This results in Θ(n²) time complexity.

**What is the “best case” for insertion sort?**

O(n).

**What is the best case time complexity of insertion sort?**

O(n).

**What is amortized cost analysis? When is it useful? Can give illustrate the idea using a simple example?**

Amortized cost analysis is a technique to analyze the average cost of a sequence of operations in an algorithm.

It's useful when an algorithm has a mix of inexpensive and occasional expensive operations, helping to provide a more balanced view of the algorithm's long-term performance.

For example, dynamic array resizing in lists ensures that even though individual resizing operations are costly, the average cost per insertion remains low, often O(1).

**Review both examples in the class notes.**

**What is the lower bound on comparison based sort algorithms?**

**What is the logic?**

**There are n! permutations. Thus there are n! leaves in the decision tree. Hence its height has to be log n!. Now log n! is Ω(n logn).**

**List two algorithms that are comparison based.**

**Lesson 5**

**Can you show merge sort is not inversion bound?**

Merge Sort is not inversion bound; it is a comparison-based sorting algorithm, and its time complexity is O(n log n) regardless of the distribution of inversions in the input data.

**What is meant by a sorting algorithm is stable?**

A stable sorting algorithm is one that preserves the relative order of equal elements in the sorted output. In other words, when two elements have the same value, a stable sorting algorithm ensures that their original order is maintained in the sorted result.

**Is Merge Sort stable (as in our class notes)?**

Merge Sort is a stable sorting algorithm.

**If not, what minor modification you can suggest so that the modified Merge sort is stable?**

**The recurrence relation and solution we already covered in Lesson 2.**

**Math Review Arithmetic Series:**

**Sum and its proof., nth term.**

S = (n/2) \* [2a + (n-1)d]

* S is the sum of the series.
* n is the number of terms in the series.
* a is the first term of the series.
* d is the common difference between the terms.

Proof for the nth Term of an Arithmetic Series:

The nth term of an arithmetic series (often denoted as "Tn") can be found using the formula:

Tn = a + (n-1)d

Where:

Tn is the nth term of the series.

a is the first term of the series.

d is the common difference between the terms.

The proof is based on the observation that each term in an arithmetic series differs from the previous term by a constant value, which is the common difference "d." Therefore, to find the nth term, you start with the first term "a" and add (n-1) times the common difference "d" to it.

For example, if you have an arithmetic series with a first term "a" of 3 and a common difference "d" of 2, you can find the 5th term (T5) as follows:

T5 = 3 + (5-1) \* 2

T5 = 3 + 4 \* 2

T5 = 3 + 8

T5 = 11

**Geometric Series: Sum (finite and infinite) and its proof. nth term.**

**finite**

S\_n = a(1 - r^n) / (1 - r)

* S\_n is the sum of the first n terms.
* a is the first term.
* r is the common ratio.
* n is the number of terms.

**infinite**

S = a / (1 - r)

* S is the sum of the infinite series.
* a is the first term.
* r is the common ratio.

Proof for the nth Term of a Geometric Series:

The nth term of a geometric series (T\_n) is calculated using the formula:

T\_n = a \* r^(n-1)

Where:

T\_n is the nth term of the series.

a is the first term.

r is the common ratio.

n is the position of the term.

The proof for the nth term is based on the observation that each term in a geometric series is obtained by multiplying the previous term by the common ratio "r." To find the nth term, you start with the first term "a" and repeatedly multiply it by "r" (n-1) times, which corresponds to the position of the term.

For example, if you have a geometric series with a first term "a" of 2 and a common ratio "r" of 3, you can find the 5th term (T5) as follows:

T5 = 2 \* 3^(5-1)

T5 = 2 \* 3^4

T5 = 2 \* 81

T5 = 162

**Mathematical Induction 数学归纳**

**Lessons after weekend**

**Radix sort**

**Data structures : Arrays, Lists, Stacks and Queues.**

**Binary search trees and BST: Traversals, search, insert, delete**