1. **Define Big Oh asymptotic notation. Prove that log n is Big Oh of n ^ {1/2}.**

To prove that log n is Big O of n^(1/2), we need to show that there exists a constant c and a value of n such that log n <= c \* n^(1/2) for all n greater than or equal to that value of n. We can start by taking the logarithm of both sides of the inequality:

log (log n) <= log (c \* n^(1/2))

Using the properties of logarithms, we can simplify this to:

log (log n) <= (1/2) \* log n + log c

Next, we can exponentiate both sides of the inequality:

log n <= exp((1/2) \* log n + log c)

Simplifying further using the laws of exponents and logarithms, we get:

log n <= c' \* sqrt(n)

where c' = exp(log c).

Since we have shown that there exists a constant c' and a value of n such that log n <= c' \* sqrt(n) for all n greater than or equal to that value of n, we have proven that log n is Big O of n^(1/2).

1. **Define inversion in a permutation. Give an array A of size n, with n distinct elements, what is the smallest and the largest number of inversions A can have?**

In mathematics, a permutation is a way of arranging a set of distinct objects in a specific order. An inversion in a permutation is a pair of elements that are in the wrong order with respect to their natural ordering. More specifically, given a permutation of the numbers 1 to n, an inversion occurs when two elements i and j, where i < j, appear in the permutation in the opposite order from their natural order, i.e., the element j appears before i in the permutation. The number of inversions in a permutation is a measure of its "disorder" or "complexity", and is used in various mathematical and algorithmic applications, such as sorting algorithms and group theory.

In an array A of size n with distinct elements, an inversion occurs when there are two elements A[i] and A[j] such that i < j but A[i] > A[j]. That is, when a smaller element appears after a larger element in the array. The smallest number of inversions that an array A can have is 0. This occurs when the array is already sorted in non-descending order, so there are no pairs of elements that need to be swapped. The largest number of inversions that an array A can have is n\*(n-1)/2. This occurs when the array is sorted in descending order, so every pair of elements needs to be swapped to sort the array in non-descending order. To see why this is true, consider that the maximum number of inversions that can occur for a given element A[i] is (n-1), since there are (n-1) elements to its right. Therefore, the maximum number of inversions that can occur in total is:

(n-1) + (n-2) + ... + 2 + 1

= n\*(n-1)/2

This is known as the triangular number formula.

1. **While expressing asymptotic time complexities whenever logarithmic factor arises, typically. we do not specify the base of the logarithm used. Give concrete argument justifying it.**

When analyzing algorithms, we use asymptotic notation to describe how their runtime grows as the input size increases. Often, the time complexity of an algorithm involves logarithmic terms, but we do not specify the base of the logarithm. This is because the choice of logarithm base does not significantly affect the overall behavior of the algorithm's runtime as the input size grows.

To see why this is the case, consider the following examples:

1. An algorithm that has a time complexity of O(log2 n) will run no more than 1 + log2 n operations for an input of size n. If we were to use a different base, say log10, we can convert the base using the change of base formula loga b = logc b / logc a. In this case, we have log10 n / log10 2 = 3.3219 \* log2 n. However, this difference in constant factor is negligible in the asymptotic sense since it is just a constant multiplier of the logarithmic term, which grows very slowly compared to the other terms.

2. Another algorithm that has a time complexity of O(log n) will run no more than log n operations for an input of size n. If we were to use a different base, say log10, we can convert the base using the change of base formula and get log10 n / log10 2. This time, the constant factor is about 3.3219 times larger than the natural logarithm base e or log2, but again, this is a constant factor and does not change the asymptotic behavior of the algorithm.

In general, the base of the logarithm is usually insignificant when it comes to asymptotic analysis. This is because the logarithm function grows very slowly compared to other functions such as polynomials, exponentials, and factorials, which are the more dominant terms in the time complexity of an algorithm. Therefore, we do not need to specify the base of the logarithm in asymptotic notation, as it does not affect the overall behavior of the algorithm's runtime as the input size grows.

1. **Consider the recurrence T(a)= T(a/5) + c, for n> 1 and T(i)= 1, where c is an absolute constant. Give best possible asymptotic upper bound on T(a). Express your answer using Big Oh asymptotic notation.**

Based on the given recurrence, we can see that T(a) = T(a/5) = T(a/25) = T(a/125) = ... = T(1) = 1. Therefore, the recurrence T(a) has log\_5(a) terms, and each term takes constant time. Hence, we can write:

T(a) = T(a/5) + c

= T(a/25) + 2c

= T(a/125) + 3c

= ...

= T(1) + log\_5(a) \* c

Therefore, we have:

T(a) = log\_5(a) \* c + 1

Using Big Oh notation, we can write:

T(a) = O(log a)

So, the best possible asymptotic upper bound on T(a) is O(log a).

1. **Consider the recurrence T(n)= T(n-1) + n for n>1 and T(1)=1. Give correct asymptotic estimate of T(n) using Big Theta notation.**

To solve this recurrence relation, we can use the iterative method of substitution. First, we expand the recurrence:

T(n) = T(n-1) + n

= T(n-2) + (n-1) + n

= T(n-3) + (n-2) + (n-1) + n

= ...

= T(1) + 2 + 3 + ... + (n-1) + n

where the last equality follows from the fact that the summation of the first n integers is n(n+1)/2. Substituting T(1)=1, we get:

T(n) = 1 + 2 + 3 + ... + (n-1) + n = n(n+1)/2

Thus, T(n) has a closed-form solution of n(n+1)/2. To obtain the asymptotic estimate of T(n), we can use Big Theta notation.

We have: n(n+1)/2 = (n^2 + n)/2

Since n^2 dominates n in the limit as n approaches infinity, we can drop the n term and write: n(n+1)/2 = Θ(n^2)

Therefore, T(n) is Θ(n^2).

1. **Suppose you are given co-ordinates cl, c2, ..... cn of n cities located on a Straight-line path. We want to build a petrol-pump on this straight-line path such that sum of the distances from cities to the petrol-pump is minimized, that is we want to find real number x such that the sum Ix-c1] + [x-c2 + .... [x-cn] is minimized. Where |a| denotes absolute value of the number a. Note that c1, c2, .. cn need not be in sorted order. Give an efficient algorithm to find x. Prove the correctness of your algorithm and analyze its time complexity.**

To solve this problem, we can use the fact that the sum of distances from the cities to the petrol-pump is a continuous function. We can use binary search to find the optimal position of the petrol-pump. Let us define f(x) as the sum of distances from the cities to the petrol-pump if the petrol-pump is located at position x. Then, we have:

f(x) = |x-c1| + |x-c2| + ... + |x-cn|

We can observe that f(x) is a convex function, as it is the sum of absolute values of linear functions. Therefore, it has a unique minimum.

To find the minimum of f(x), we can use binary search. Let l and r be the left and right endpoints of the interval on which we want to find the minimum, respectively. Initially, we set l = min(c1, c2, ..., cn) and r = max(c1, c2, ..., cn).

At each step of the binary search, we compute the midpoint m = (l+r)/2, and evaluate f(m) and f(m+1). If f(m) < f(m+1), then the minimum must lie in the interval [l, m], so we update r = m. Otherwise, the minimum must lie in the interval [m+1, r], so we update l = m+1. We repeat this until the interval [l, r] becomes sufficiently small.

The correctness of this algorithm follows from the fact that f(x) is a convex function and has a unique minimum. At each step of the binary search, we are reducing the search interval by half, so the algorithm takes O(log n) iterations. Evaluating f(x) for a given x takes O(n) time, so the overall time complexity of the algorithm is O(n log n).

Algorithm:

1. Let l = min(c1, c2, ..., cn) and r = max(c1, c2, ..., cn).

2. Repeat until the interval [l, r] becomes sufficiently small:

a. Let m = (l+r)/2.

b. Evaluate f(m) and f(m+1).

c. If f(m) < f(m+1), set r = m. Otherwise, set l = m+1.

3. Return the midpoint of the final interval [l, r] as the optimal position of the petrol-pump.

Note that the "sufficiently small" condition for terminating the loop can be chosen based on the required accuracy of the solution. For example, we can terminate the loop when the length of the interval [l, r] becomes less than a specified value.

1. **Consider Josephus game with N players. Players are numbered from 1 to N. Players 1 to N are standing round a circle in clockwise direction with the first player holding a sword. Player 1 kills player 2 and passes the sword to player 3. Player 3 kills player 4 and passes sword to player 5. This goes on. Any player having the sword kills the next alive person in clockwise direction and passes the sword to the next alive person in clockwise direction. The game continues till only one player is left, and this player is called the winner of the game. Who is the winner for Josephus game with N= 2 ^ {2 ^ 100} + 2 ^ 100 - 1 many players?**

The Josephus game is a famous problem in computer science and mathematics. The solution for this game is a recursive algorithm and can be solved using mathematical formulas.

Let's first consider the case of a power of 2, i.e., N=2^k players. In this case, the winner of the game is always the player with number 1. To see why, let's denote the players as P1, P2, ..., PN in clockwise order. After the first round, the players who are left are P1, P3, P5, ..., PN-1. In the second round, P1 kills P3, and the remaining players are P5, P7, ..., PN-1. This pattern continues until there is only one player left, which is always P1.

Now, let's consider the general case of N players. We can write N as N = 2^k + m, where k is the largest integer such that 2^k <= N, and m is the remainder. We can then find the winner of the game as follows:

1. Play the Josephus game with the first 2^k players, which we know from the above is player 1.

2. The remaining m players are numbered from 2^k+1 to N. We can then relabel the players as follows: player 2^k+1 becomes player 1, player 2^k+2 becomes player 2, and so on up to player N, who becomes player m. We can then play the Josephus game again with these m players, which we can denote by J(m).

3. The winner of the game is the player who survives in the game J(m).

So, to find the winner of the game with N=2^(2^100) + 2^100 - 1 players, we need to find k and m, and then compute J(m). We have:

k = 2^100

2^k = 2^(2^100)

2^k + m = N = 2^(2^100) + 2^100 - 1

m = 2^100 - 1

So, we need to find the winner of the Josephus game J(2^100 - 1) with 2^100 - 1 players. Using the above formula for power of 2, we can find that the winner of J(2^100 - 1) is player 2. Therefore, the winner of the original game with N=2^(2^100) + 2^100 - 1 players is player 2^(2^100) + 2^100.

1. **Consider a chocolate bar of size m x n your task is to divide it in pieces of size 1x 1 by cutting the bar, Every valid move involves cutting a single piece along any one grid line. (E.g. suppose you have bar of size 3 x 4 to begin with, then a single horizontal cut at y =1 grid line results into two pieces of dimensions 1 x 4, 2 x 4 respectively). Note that in any move you are only allowed to cut a single piece. What is the minimum number of moves required for this task?**

To divide a chocolate bar of size m x n into pieces of size 1 x 1, we need to make (m-1) vertical cuts and (n-1) horizontal cuts.

Therefore, the minimum number of moves required to divide the chocolate bar into pieces of size 1 x 1 is:

Minimum number of moves = (m-1) + (n-1) = m + n - 2

For example, if the chocolate bar is of size 3 x 4, then the minimum number of moves required would be:

Minimum number of moves = (3-1) + (4-1) = 2 + 3 - 2 = 3

We need to make two vertical cuts and two horizontal cuts to divide the chocolate bar into pieces of size 1 x 1.

1. **What is the remainder you will get when you divide 11 ^ 58 by 59? Justify?**

To find the remainder when $11^{58}$ is divided by 59, we can use Fermat's Little Theorem, which states that if $p$ is a prime number and $a$ is an integer not divisible by $p$, then $a^{p-1}\equiv 1\pmod{p}$. In this case, $p=59$ is a prime number and $11$ is not divisible by $59$, so we have: $11^{58}\equiv 11^{59-1}\equiv 1\pmod{59}$

Therefore, the remainder when $11^{58}$ is divided by 59 is 1.

1. **Compare divide and conquer and Dynamic Programming strategies. For each of the strategies give an example of a computational problem where the strategy can be effectively applied.**

Both Divide and Conquer and Dynamic Programming are well-known algorithmic techniques used in computer science to solve complex problems.

Divide and Conquer involves breaking down a problem into smaller sub-problems, solving each sub-problem independently, and then combining the solutions to form the final solution to the original problem. It is typically used when the problem can be easily divided into smaller sub-problems, and the sub-problems are independent of each other. Some examples of problems that can be effectively solved using Divide and Conquer are:

1. Merge Sort: Sorting a large array can be done by dividing it into smaller arrays, sorting each sub-array recursively using merge sort, and then merging the sorted sub-arrays.

2. Binary Search: Searching for an element in a sorted array can be done by dividing the array into halves, and recursively searching in the appropriate half.

Dynamic Programming involves breaking down a problem into smaller sub-problems, solving each sub-problem only once, and storing the solution to each sub-problem in memory. This way, the solution to a larger problem can be computed by combining the solutions to its smaller sub-problems. It is typically used when the problem can be divided into overlapping sub-problems, and the sub-problems can be solved using previously computed solutions. Some examples of problems that can be effectively solved using Dynamic Programming are:

1. Fibonacci series: Computing the nth term in the Fibonacci series can be done using Dynamic Programming, by storing the solution to each term in an array, and using the stored solutions to compute the next term.

2. Longest Common Subsequence: Finding the longest common subsequence of two strings can be done using Dynamic Programming, by breaking down the problem into sub-problems of finding the longest common subsequence of smaller substrings, and storing the solutions to each sub-problem in memory.

In summary, Divide and Conquer is typically used for problems that can be easily divided into independent sub-problems, while Dynamic Programming is used for problems that can be divided into overlapping sub-problems, and the sub-problems can be solved using previously computed solutions.

1. **If P=NP prove that NP=CoNP**

If P = NP, then NP = CoNP. This can be shown as follows:

If P = NP, then every problem in NP can be solved in polynomial time. In other words, given a solution to a problem in NP, we can check that solution in polynomial time.

Now consider the complement of a problem in NP. If the original problem can be solved in polynomial time, then its complement can also be solved in polynomial time, since we can simply run the polynomial time algorithm on the input and return the opposite of the output.

Therefore, every problem in NP has a polynomial time algorithm to solve its complement, and thus NP = CoNP.

Note that the converse is not necessarily true; just because NP = CoNP doesn't necessarily mean that P = NP.

1. **Let A be n x n array of integers such that the numbers in every row and column of A are in sorted order. On input an integer a, give an efficient algorithm to search a in A. Give proper pseudocode and brief argument justifying correctness of the algorithm. Analyze the time complexity of the algorithm.**

To search for an integer `a` in the `n x n` array `A` of integers, we can use the following algorithm:

def search\_in\_sorted\_array(A, a):

n = len(A)

i = 0 # starting row

j = n-1 # starting column

while i < n and j >= 0:

if A[i][j] == a:

return True

elif A[i][j] > a:

j -= 1

else:

i += 1

return False

The algorithm starts by initializing `i` and `j` to the starting positions of the bottom-left element of the array, which is the smallest element in the last column. It then enters a loop that continues until it either finds the element `a` or reaches the end of the array. In each iteration of the loop, it checks the value of the element at the current position `(i,j)` of the array. If this value is equal to `a`, the algorithm returns `True`. Otherwise, if the value is greater than `a`, the algorithm moves to the left by decrementing `j`. If the value is less than `a`, the algorithm moves up by incrementing `i`. If the loop completes without finding the element `a`, the algorithm returns `False`.

The correctness of the algorithm can be justified as follows: Since the rows and columns of `A` are sorted in ascending order, if the algorithm starts at the bottom-left element and moves up and to the right, it will always encounter elements that are greater than or equal to the current element. If it encounters an element equal to `a`, it returns `True`. If it encounters an element greater than `a`, it moves left to find a smaller element. If it encounters an element less than `a`, it moves up to find a larger element. Therefore, the algorithm will eventually either find the element `a` or reach the end of the array without finding it.

The time complexity of the algorithm is `O(n)` since it performs a constant number of operations `n` times at most (i.e., until it reaches the top-right corner of the array).

1. **Compare Las Vegas and Monte Carlo algorithms. Give an example of each**.

Las Vegas and Monte Carlo algorithms are both probabilistic algorithms used in computer science and mathematics to solve problems. However, they differ in the way they guarantee their output.

Las Vegas algorithms are designed to give the correct output with a certain probability. These algorithms have a random component that affects their output, and they can either terminate with the correct result or indicate that they failed to find a solution. An example of a Las Vegas algorithm is the randomized quicksort algorithm, which has an expected running time of O(n log n) but can run in O(n^2) in the worst case.

Monte Carlo algorithms, on the other hand, are designed to give an approximate solution with a certain level of confidence. These algorithms use random sampling to estimate the solution, and the accuracy of the output depends on the number of samples used. An example of a Monte Carlo algorithm is the Monte Carlo method for approximating pi, which involves randomly generating points within a square and calculating the ratio of points inside a circle inscribed in the square.

In summary, Las Vegas algorithms aim to guarantee correctness with a certain probability, while Monte Carlo algorithms aim to provide approximate solutions with a certain level of confidence.

Example of Las Vegas algorithm: Randomized quicksort algorithm.

Example of Monte Carlo algorithm: Monte Carlo method for approximating pi.

1. **Give prover-verifier based definitions of complexity class NP and give concrete examples of problems characterizing NP class.**

To give prover-verifier based definitions of the complexity class NP, we first need to define the concept of a certificate or proof. A certificate is a string that can be used to verify that a solution to a problem is correct. The prover is a hypothetical entity that knows a correct solution to the problem, and it sends the certificate to the verifier, which can efficiently verify that the solution is indeed correct.

With this in mind, we can define the class NP as follows: A decision problem L is in NP if there exists a polynomial-time algorithm V and a polynomial p such that:

1. For every instance x of L, there exists a certificate y of length p(|x|) such that V(x,y) = 1 if and only if x is in L.

2. V runs in polynomial time in the length of x, i.e., V runs in time O(p(|x|)).

In other words, a problem is in NP if it is possible to verify the correctness of a solution in polynomial time with the help of a certificate, even if finding the solution itself is difficult. The prover is allowed to take exponential time to find the certificate, but the verifier must run in polynomial time.

Examples of problems in NP include:

- Boolean satisfiability (SAT): Given a Boolean formula in conjunctive normal form (CNF), is there an assignment of truth values to the variables that makes the formula true?

- Traveling salesman problem (TSP): Given a set of cities and the distances between them, is there a tour that visits each city exactly once and has total length at most k?

- Subset sum: Given a set of integers and a target sum, is there a subset of the integers that adds up to the target sum?

- Knapsack problem: Given a set of items, each with a weight and a value, and a maximum weight capacity for a knapsack, what is the most valuable subset of items that can be put into the knapsack without exceeding its capacity?

1. **We say that a Boolean formula F (Over Boolean variables x1, x2, …, xn and Boolean operators AND, OR, NOT) is Tautology if formula F evaluates to TRUE for all Boolean assignments for the variables, Let P be the problem of testing if given Boolean formula F is a tautology or not. The Yes instance of the problem is when the formula F is a Tautology. Which of the following statement(s) is/are true. Justify your answer,**

**1. Problem P is in complexity class NP.**

**2. Problem P is in complexity class Co-NP.**

The problem of testing if a Boolean formula is a tautology or not is in the complexity class co-NP.

To see why, note that to show that a formula F is a tautology, one has to show that it is not satisfiable (i.e., there does not exist any assignment of variables that makes F false). This can be checked in polynomial time by simply checking all possible assignments of variables and verifying that F is true for all of them.

Therefore, P is in co-NP, since the complement of the problem (i.e., testing whether a formula is satisfiable or not) is in NP.

It is worth noting that some researchers believe that the problem of testing tautology is in fact in the complexity class PSPACE, which is a superset of NP and co-NP. This is because the problem can be solved using brute force, which takes exponential time in the worst case, but only polynomial space. However, this is still an open question in complexity theory.

1. **Prove that any comparison based sorting algorithm must perform Omega(n log n) comparisons to sort a set of n numbers, where Omega denotes Big-Omega asymptotic notation.**

To prove that any comparison-based sorting algorithm must perform Omega(n log n) comparisons to sort a set of n numbers, we can use a technique called decision trees.

A decision tree is a binary tree that represents all possible comparisons made by a sorting algorithm. Each internal node in the tree represents a comparison between two elements, and the two child nodes represent the two possible outcomes of the comparison. The leaves of the tree represent the sorted permutations of the input set.

Since the algorithm must produce the correct output for any input permutation, the decision tree must have at least n! leaves, which is the number of possible permutations of n elements.

Now, let's assume that our sorting algorithm performs at most k comparisons for any input of size n. Then, the decision tree for this algorithm has at most 2^k leaves, since each comparison has two possible outcomes.

Since we need a decision tree with at least n! leaves to represent all possible permutations, we have:

n! <= 2^k

Taking the logarithm of both sides, we get:

log n! <= k log 2

Using Stirling's approximation for n!, we have:

n log n - n <= k log 2

Rearranging the terms, we get:

k >= (n log n - n) / log 2

As n approaches infinity, the constant term -n becomes negligible, and we have:

k >= n log n / log 2

Therefore, any comparison-based sorting algorithm must perform at least Omega(n log n) comparisons in the worst case.

1. **Consider the standard algorithm to compute maximum element of array A[1-n]. We start with initializing 4 variable max with A[1] and do a pass on entire array A[2-n], in each step we compare max with A[i], if A[i] is larger than max, we update max to A[i]. What is expected number of times the variable max gets updated if array A consists of a permutation over 1,2, 3, ..n chosen uniformly at random.**

Let X\_i be a random variable which takes value 1 if the ith element of array A is larger than the current max, and 0 otherwise. Then, the number of updates to the variable max is equal to the sum of all X\_i's.

The probability that the ith element is larger than the current max is 1/i, since there are i possible values in the array A that could be the largest. Therefore, E[X\_i] = 1/i.

By linearity of expectation, the expected number of updates to the variable max is:

E[sum(X\_i)] = sum(E[X\_i]) from i = 2 to n

= sum(1/i) from i = 2 to n

This sum can be estimated using the harmonic series. It is well known that the harmonic series diverges, so this sum is unbounded as n approaches infinity. Therefore, the expected number of updates to the variable max grows without bound as the size of the array A increases.

In other words, as the size of the array A becomes very large, we expect to see a large number of updates to the variable max, regardless of how the elements are arranged.

1. **Prove the following property of Huffman encoding scheme. If some character occurs with frequency strictly more than 2/5, then there is guaranteed to be a codeword of length 1.**

Let's assume that there exists a character that occurs with frequency strictly more than 2/5 in the input sequence, but there is no codeword of length 1 for that character in the Huffman encoding scheme.

Since there is no codeword of length 1 for this character, it must have a minimum frequency of at least 1/2, because any character with a frequency less than 1/2 can be represented with a codeword of length 1.

Let's consider a subsequence of the input that consists of this character and all other characters that have a frequency less than 1/2. The total frequency of this subsequence is at most 1/2 + (n-1)/2 = n/2, where n is the length of the input sequence.

Since the character we're interested in occurs with frequency strictly more than 2/5, its frequency in this subsequence must be at least 1/2 - (n-1)/2 = (3-n)/2.

Let's consider the Huffman tree for this subsequence. The two nodes with the highest frequencies must correspond to this character and one other character with frequency less than 1/2. Since the frequency of the other character is less than 1/2, it can be represented with a codeword of length 1, which means that the codeword for the character we're interested in must have length at least 2.

However, this means that the average codeword length for this subsequence is at least (1/2)\*(2) + [(n-1)/2]\*(1) = (n+1)/2, which is greater than the entropy of the subsequence.

This contradicts the optimality of the Huffman encoding scheme, which guarantees that the average codeword length is equal to the entropy. Therefore, our assumption that there exists a character that occurs with frequency strictly more than 2/5 but has no codeword of length 1 in the Huffman encoding scheme must be false. Hence, it is proven that if some character occurs with frequency strictly more than 2/5, then there is guaranteed to be a codeword of length 1.

Let n be an odd natural number and Ay 75s a sored array of egers such that