INFO 6205 – Program Structures and Algorithms
Assignment 5

Student Name: Yuetong Guo

Professor: Nik Bear Brown

Instructions:

- Support your approach with algorithm and run through an example if possible.
- Write pseudo codes wherever possible
- Include code files that have screenshots attached alongside them.

Q1 (5 Points)

A mobile game randomly and uniformly awards a special coin for completing each level. There are n different types of coins. Assuming all levels are equally likely to award each coin, how many levels must you complete before you expect to have ≥ 1 coin of each type?

Ans:

The problem is a variation of the well-known "coupon collector" problem. In this scenario, you're trying to collect n distinct coins, where each coin has an equal probability of being awarded after completing a level. The expected number of levels to complete to collect at least one coin of each type can be calculated using the concept of expected waiting time.

The answer to this question is given by the formula:

$$E(X) = n * (1 + 1/2 + 1/3 + ... + 1/n)$$

where E(X) is the expected number of coupons needed to obtain a complete set, and the sum 1 + 1/2 + 1/3 + ... + 1/n is called the harmonic series.

As you can see from the formula, the expected number of coupons needed grows with the number of distinct coupons n and is roughly proportional to n*log(n).

So the expected number of steps is O(n*log n).

Let E[Xj] denote the expected waiting time to obtain a new coin type after having j distinct coins. The probability of getting a new coin type is (n-j)/n. Hence, the expected waiting time is n/(n-j).

The total expected waiting time E[X] is the sum of the individual waiting times:

$$E[X] = \Sigma\ E[Xj] \text{ from } j=0 \text{ to } n-1$$

$$E[X] = \Sigma\ (n/(n-j)) \text{ from } j=0 \text{ to } n-1$$

This sum can be written in terms of the harmonic series, H(n):

$$E[X] = n * H(n)$$

H(n) is the sum of the reciprocals of the first n natural numbers:

$$H(n) = 1 + 1/2 + 1/3 + ... + 1/n$$

H(n) is approximately equal to ln(n+1), which gives us an asymptotic bound for E[X]:

$$\ln(n+1) < H(n) < 1 + \ln n$$

Thus, the expected number of levels to complete to collect all n distinct coins is approximately n * H(n), where H(n) is the harmonic series.

Algorithm to estimate levels for all coin types:

1. Initialize n (number of distinct coins), E[X] (expected waiting time) to 0

2. For i = 1 to n, increment E[X] by n/i

3. Return E[X]

The pseudo code is:

```
def estimate_levels_to_collect_all_coins(n):
    expected_waiting_time = 0
    for i in range(1, n+1):
        expected_waiting_time += n / i
    return expected_waiting_time

n = <number_of_distinct_coins>
result = estimate_levels_to_collect_all_coins(n)

```

Q2 (10 Points)

PushPush is a 2-D pushing-blocks game with the following rules:

1. Initially, the planar square grid is filled with some unit-square blocks (each occupying a cell of the grid) and the robot placed in one cell of the grid.
2. The robot can move to any adjacent free square (without a block).
3. The robot can push any adjacent block, and that block slides in that direction to the maximal extent possible, i.e., until the block is against another block.
4. Pushing means moving away (farther) from the robot (not pulling closer to robot)
5. The goal is to get the robot to a particular position.

A solution to PushPush is specified as a list of the following moves and x, y coordinates:

```
MoveRobot(x,y)  # moves the robot from its current position to the position x, y
PushBlock(x,y)  # pushes a block from its current position to the position x, y
CheckGoal(robot)  # takes the position of the robot and checks whether it is at the goal
```

Is the Push Push problem in NP? If so prove it.

Ans:

To demonstrate that the PushPush problem is in NP, we must show that given a potential solution, we can verify its correctness in polynomial time. Let's assume we are provided with a solution, which is a sequence of moves, such as MoveRobot(x1,y1), PushBlock(x2,y2), ..., CheckGoal(xn,yn), with (xn,yn) being the robot's final position.

To verify this solution, we can simulate the moves on the initial state of the game and check whether the final robot position matches the given goal position, **the algorithms design could be**:

1. Start with the initial state of the game (including block layout and initial robot position).

2. For each MoveRobot(x,y) move, ensure (x,y) is a valid adjacent free square and move the robot to (x,y).

3. For each PushBlock(x,y) move, ensure (x,y) is a valid adjacent block, and check if (x',y') is a valid maximal adjacent free square. If valid, move the block to (x',y').

4. After simulating all moves, use CheckGoal(x,y) to confirm if the robot's final position matches the goal position.

This process has a polynomial-time complexity relative to the input size (number of blocks and grid dimensions) since each move can be verified in constant time.

Therefore, the PushPush problem is in NP.

Pseudo code for the verification process:

```
def is_solution_valid(initial_state, moves, goal_position):
    game_state = initial_state.copy()

    for move in moves:
        if move.type == "MoveRobot":
            if is_valid_adjacent_free_square(game_state, move.x, move.y):
                game_state.move_robot(move.x, move.y)
            else:
                return False

        elif move.type == "PushBlock":
            if is_valid_adjacent_block(game_state, move.x, move.y):
                x_prime, y_prime = find_maximal_extent(game_state, move.x, move.y)
                if x_prime is not None:
                    game_state.move_block(move.x, move.y, x_prime, y_prime)
                else:
                    return False
            else:
                return False

    return check_goal(game_state.robot_position, goal_position)

```

This verification process can be executed in polynomial time, confirming that the PushPush problem is in NP.

Q3 (5 Points)

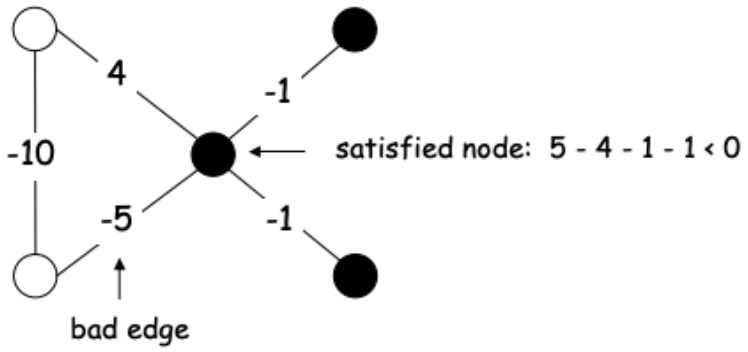
What makes a configuration “stable” in the Hopfield Neural Network algorithm?

Ans:

A configuration in a Hopfield Neural Network is considered stable if all nodes are satisfied. In other words, each node's state does not change after an update in the network. To further clarify the stability concept, let's define a few terms related to configurations and nodes:

Good edge:
With respect to a configuration S, an edge e = (u,v) is considered good if $W_e * S_u * S_v < 0$.
This means that if $W_e < 0$, then $S_u = S_v$; and if $W_e > 0$, $S_u \neq S_v$.

Satisfied node:
With respect to a configuration S, a node u is satisfied if the weight of incident good edges is greater than or equal to the weight of incident bad edges.



Mathematically, this can be expressed as:

$$\Sigma\ (v: e=(u,v) \in E)\ W_e * S_u * S_v <= 0$$

In summary, a configuration in a Hopfield Neural Network is stable when all nodes are satisfied, and their states do not change after any updates in the network. This is determined by the balance between the weights of good and bad edges connected to each node.

Q4 (5 Points)

Does the payoff matrix below have any Nash equilibria? Why or why not?

	Cooperate	Defect
Cooperate	2,1	1,2
Defect	1,2	2,1

Ans:

To determine if there are any Nash equilibria in the given payoff matrix, we must examine each action profile and check if both players have no incentive to deviate unilaterally from their current strategy.

The check procedure should be:

Step 1: Determine the row player's best response to each of the column's possible strategies.

Step 2: Determine the column player's best response to each of the row's possible strategies.

Step 3: Any outcome that is a best response for both players is a Nash equilibrium.

For the matrix in this question, follow above rule we go through and has:

1. (Cooperate, Cooperate)

   Player 2 can increase its payoff from 1 to 2 by defecting instead of cooperating. Thus, this action profile is not a Nash equilibrium.

2. (Cooperate, Defect)

   Player 1 can increase its payoff from 1 to 2 by defecting instead of cooperating. Thus, this action profile is not a Nash equilibrium.

3. (Defect, Cooperate)

   Player 1 can increase its payoff from 1 to 2 by cooperating instead of defecting. Thus, this action profile is not a Nash equilibrium.

4. (Defect, Defect)

   Player 2 can increase its payoff from 1 to 2 by cooperating instead of defecting. Thus, this action profile is not a Nash equilibrium.

After examining all action profiles, we conclude that the game has no Nash equilibrium because there is no action profile where both players have no incentive to unilaterally change their strategies.

Q5 (5 Points)

Coin is heads with probability p and tails with probability 1-p. How many independent flips X until first heads? (Express the expectation as a function of p.)

Ans:

Let X be a discrete random variable representing the number of flips needed to obtain the first heads. The probability of getting the first heads after i flips is given by the geometric distribution:

$$P(X = i) = (1-p)^{(i-1)} * p, \text{ for } i = 1, 2, 3, ...$$

The expected value of X is:

$$E[X] = \sum(\text{sum from } i=1 \text{ to } \infty)\ i * P(X = i) = \sum(\text{sum from } i=1 \text{ to } \infty)\ i * (1-p)^{(i-1)} * p$$

Now, we can rewrite the sum as:

$$E[X] = p * \sum(\text{sum from } i=1 \text{ to } \infty)\ i * (1-p)^{(i-1)}$$

Using the known formula for the expected value of a geometric distribution:

$$E[X] = 1/p$$

This formula gives the expected number of flips until the first heads as a function of p.

Q6 (5 Points)

Give an argument or proof for the questions below:

A) Can always convert a Las Vegas algorithm into a Monte Carlo algorithm?

Ans:

A. Yes, we can always convert a Las Vegas algorithm into a Monte Carlo algorithm.

A Las Vegas algorithm is a type of randomized algorithm that always produces the correct answer, but its running time is variable and depends on random choices made during its execution. On the other hand, a Monte Carlo algorithm is a randomized algorithm that has a fixed running time, but its correctness depends on the probability of getting the right answer.

To convert a Las Vegas algorithm into a Monte Carlo algorithm, we can impose a time limit on the execution of the Las Vegas algorithm. If the algorithm does not find a solution within the time limit, we stop it and return the best solution found so far. The resulting Monte Carlo algorithm will have a fixed running time, while the correctness will depend on the probability of finding the correct solution within the given time limit.

B) Can always convert a Monte Carlo algorithm into a Las Vegas algorithm?

Ans:

B. No, we cannot always convert a Monte Carlo algorithm into a Las Vegas algorithm.

A Monte Carlo algorithm has a fixed running time and a probability of providing the correct answer, whereas a Las Vegas algorithm always provides the correct answer, but its running time is variable.

To convert a Monte Carlo algorithm into a Las Vegas algorithm, we would need to modify the algorithm to ensure that it always provides the correct answer while maintaining a variable running time. However, this may not always be possible, especially for problems that are hard to solve deterministically in polynomial time (e.g., NP-complete problems).

For example, consider the problem of determining whether a given Boolean formula is satisfiable. A Monte Carlo algorithm for this problem might randomly assign truth values to each variable in the formula and then check if the formula is satisfied. If the algorithm returns "yes," there is a high probability that the formula is satisfiable, but if it returns "no," we cannot be certain that the formula is unsatisfiable. However, there is no known deterministic algorithm that always solves this problem in polynomial time, and it is not guaranteed that we can convert the Monte Carlo algorithm into a Las Vegas algorithm that always gives the correct answer while maintaining a variable running time.

Q7 (5 Points)

Euclid's algorithm is an efficient method for computing the greatest common divisor (GCD) of two numbers, the largest number that divides both of them without leaving a remainder.
The algorithm in pseudocode is below:

```
int euclids_algorithm(int m, int n){
  if(n == 0)
     return m;
   else
     return euclids_algorithm(n, m % n);}
```

A) Write a recurrence relation for Euclid's algorithm.

Ans:

The recurrence relation for Euclid's algorithm is T(n) = T(n mod m) + c, where m is roughly n/2.

Since this is of the same form as T(n) = T(n/2) + c, the time complexity of Euclid's algorithm is O(log n).

The recurrence relation T(n) = T(n/b) + c corresponds to a divide-and-conquer algorithm that divides the input into b subproblems of size n/b, and has a time complexity of O(log_b n).

The inequality T(n) ≤ T(n/2) + O(1) is an upper bound on the time complexity of algorithms that divide the input in half at each level of recursion, but it is not tight enough to capture the time complexity of Euclid's algorithm.

B) Give an expression for the runtime T(n) if your Euclid's algorithm recurrence can be solved with the Master Theorem.

Ans:

Using the Master Theorem, we can solve the recurrence T(n) ≤ T(n/2) + O(1). The Master Theorem applies to recurrences of the form:

T(n) = a*T(n/b) + f(n), where a >= 1 and b > 1

For Euclid's Algorithm, we have a = 1, b = 2, and f(n) = O(1). There are three cases in the Master Theorem:

If f(n) = Θ(n^c), where c < log_b(a), then T(n) = Θ(n^(log_b(a)))
If f(n) = Θ(n^c), where c = log_b(a), then T(n) = Θ(n^c * log^(p+1)(n)), where p is some constant
If f(n) = Θ(n^c), where c > log_b(a), then T(n) = Θ(f(n))
In this case, log_2(1) = 0, and f(n) = n^0 * c, which is equal to case 2. Therefore, the runtime T(n) can be expressed as:

T(n) = Θ(n^c * log^(p+1)(n)) = Θ(n^0 * log^(0+1)(n)) = Θ(log n)

Alternatively, if you used the recurrence T(n) = T(n/b) + c, the runtime would be:

T(n) = Θ(n^c * log_b^(p+1)(n)) = Θ(n^0 * log_b^(0+1)(n)) = Θ(log_b(n))

Both expressions are correct, as they describe the runtime complexity of Euclid's algorithm in terms of the input size n.

Q8 (10 Points)

Consider a random algorithm to find a maximum Independent Set in a graph. That is, a graph $G = (V, E)$ with a node having the value 0 or 1 and an edge joining pairs of nodes. Two nodes are in conflict if they belong to the same set (i.e. a 0 node connecting to a 0 node or a 1 node connecting to a 1 node). We know finding the maximum-size Independent Set *S,* is NP- Complete so we want to find as large an Independent Set *S* as we can using a randomized algorithm. We will suppose for purposes of this question that each node in has exactly *d* neighbors in the graph *G.* (That is, each node is in conflict with exactly *d* other nodes.)

Consider the following randomized algorithm to find a large Independent Set in this special graph. *Each node P_i independently picks a random value x_i; it sets x_i to* 1 *with probability p=0.5 and sets x_i to* 0 *with probability* 1 – *p=0.5. It then decides to enter the set S if and only if it chooses the value* 1, *and each of the d nodes with which it is connected chooses the value* 0. Give a formula for the expected size of *S* when *p* is set to 0.5.

Ans:

To find the expected size of the Independent Set S in this special graph, we first need to understand the conditions for a node to be included in the set S. A node Pi will be included in S if it picks the value 1 with a probability of 0.5 and all its d neighbors pick the value 0 with a probability of 0.5 each. The selection process is independent for each node.

Algorithm:

1. For each node i in graph G, do the following:

    a. Generate a random number r in [0, 1].

    b. If r <= 0.5, set xi to 1; otherwise, set xi to 0.

    c. If xi is 1 and all d neighbors have the value 0, add node i to the set S.

2. Return the set S.

Now, we can compute the probability of a node Pi being included in the set S. Since all its d neighbors must have the value 0, and the probability of a node choosing 0 is 0.5, we get:

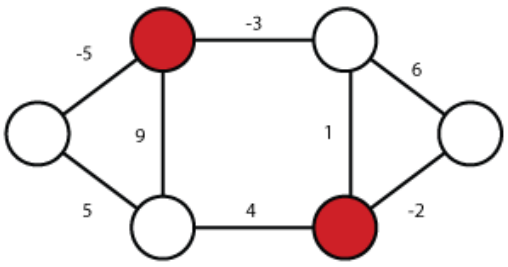
$$P[Pi\ selected] = (1/2) * (1/2)^d = (1/2)^{(d+1)}$$

Considering that there are n nodes in the graph, and the selection process for each node is independent, we can use the linearity of expectation to compute the expected size of the set S:

$$Expected\ size\ of\ S = n * (1/2)^{(d+1)}$$

Q9 (5 Points)

Use the *Hopfield Neural Network State-flipping algorithm* to find a stable configuration on the weighted undirected network graph above. Show your work. Draw the *final* stable configuration.



Ans:

We'll use the Hopfield Neural Network State-flipping algorithm to find a stable configuration for the given graph. We'll using the labeling scheme A-F clockwise from far left as below and considering red as -1 and white as 1.

Definitions:

1. An edge e = (u, v) is good with respect to a configuration S if w_e * s_u * s_v < 0.
   That means if w_e < 0, then s_u = s_v; if w_e > 0, s_u != s_v.
2. A node u is satisfied with respect to a configuration S if the weight of incident good edges >= the weight of incident bad edges.
3. A configuration is stable if all nodes are satisfied.

Algorithm:

1. Start with an arbitrary configuration S.
2. While the current configuration is not stable:
   a. Choose an unsatisfied node u.
   b. Flip the state of node u (s_u = -s_u).
3. Return the stable configuration S.

Applying the algorithm to the given graph:

A: 5 + 5 = 10 > 0, flip (A becomes red)

A: -5 - 5 = -10 <= 0, don't flip

B: -5 - 9 + 3 = -11 <= 0, don't flip

C: 3 + 6 - 1 = 8 > 0, flip (C becomes red)

C: -3 - 6 + 1 = -8 <= 0, don't flip

Recheck B, D, and E, as they are incident to C:

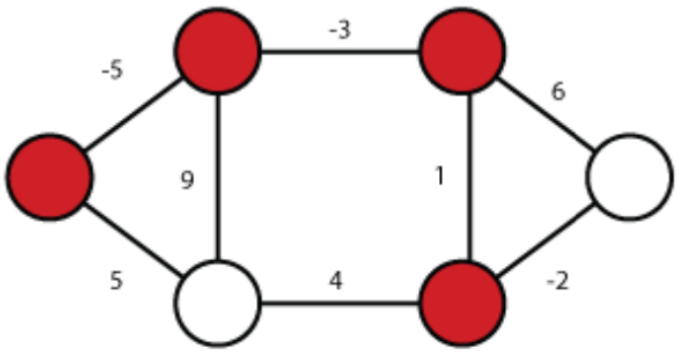
B: -5 - 9 - 3 = -17 <= 0, don't flip

D: -6 + 2 = -4 <= 0, don't flip

E: 2 + 1 - 4 = -1 <= 0, don't flip

F: -5 - 9 - 4 = -18 <= 0, don't flip

All nodes are satisfied, and the final stable configuration is:



Q10 (5 Points)

A) Sort the list of integers below using Quicksort. Show your work. (22, 13, 29, 1, 2, 31, 33, 11)

Ans:

I'll use the last element as the pivot for each partition and perform in-place swaps.
The pseudocode is as below:

```
def quicksort(arr, low, high):
    if low < high:
        pivot_location = partition(arr, low, high)
        quicksort(arr, low, pivot_location - 1)
        quicksort(arr, pivot_location + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    left_wall = low - 1

    for i in range(low, high):
        if arr[i] < pivot:
            left_wall += 1
            arr[left_wall], arr[i] = arr[i], arr[left_wall]

    arr[left_wall + 1], arr[high] = arr[high], arr[left_wall + 1]
    return left_wall + 1
```

Now, let's run through the Quicksort algorithm using the provided list:

Initial List: (22, 13, 29, 1, 2, 31, 33, 11)

First Partition using pivot 11:
(1, 2, 11) + (22, 13, 29, 31, 33)

Recursively partition the first sublist:
(1) + (2, 11) + (22, 13, 29, 31, 33)

Recursively partition the second sublist:
(1) + (2, 11) + (13, 22, 29, 31, 33)

Recursively partition the third sublist:
(1) + (2, 11) + (13) + (22, 29, 31, 33)

Recursively partition the fourth sublist:
(1) + (2, 11) + (13) + (22) + (29, 31, 33)

Recursively partition the fifth sublist:
(1) + (2, 11) + (13) + (22) + (29) + (31, 33)

Recursively partition the sixth sublist:
(1) + (2, 11) + (13) + (22) + (29) + (31) + (33)

Sorted List: (1, 2, 11, 13, 22, 29, 31, 33)

B) Write a worst case and average case recurrence relation for Quicksort.

Ans:

Worst case performance (big-O): $O(n^2)$

Average case performance (big-O): O(n log n)

Recurrence Relations:

Worst-case recurrence relation: T(n) = T(n-1) + O(n)

Average-case recurrence relation: T(n) = 2T(n/2) + O(n)

C) Give an expression for the runtime T(n) if your worst case and average case Quicksort recurrence relations can be solved with the Master Theorem.

Ans:

Runtime expression using Master Theorem:

For the average-case recurrence relation, we have a = 2, b = 2, and f(n) = O(n).

Therefore, log_b(a) = log_2(2) = 1 and c = n^1, which corresponds to case 2 of the Master Theorem.

Applying the Master Theorem for the average case, we get:

T(n) = Θ(n^(log_b(a)) * log(p+1)*n) = Θ(n^1 * log(0+1)*n) = Θ(n * log n)

Hence, the average-case runtime T(n) for Quicksort is Θ(n * log n).

Q11 (10 Points)

Ebay is considering a very simple online auction system that works as follows. For an auction, if there are *n bidders*; agent *i* has a bid b_i, which is a positive natural number. We will assume that all bids b_i are distinct from one another. The bidders appear in an order chosen uniformly at random, each proposes its bid b_i in turn, and at all times the system maintains a variable b^* equal to the highest bid seen so far. (Initially b^* is set to 0.) In essence, this system collects all n bids then presents them uniformly at random.

What is the expected number of times that b^* is updated when this process is executed, as a function of the parameters in the problem?

Example： Suppose $b_1 = 22$, $b_2 = 33$, and $b_3 = 11$, and the bidders arrive in the order 1, 3, 2. Then b^* is updated for 1 (b_0 is initially 0 so first bid is always updated) and 2, but not for 3.

Ans:

Let X be a random variable representing the number of times that b* (the highest bid) is updated. Let X_i be an indicator random variable such that X_i = 1 if the i-th bid in the order causes b* to be updated and X_i = 0 otherwise. We can represent X as the sum of these indicator random variables: X = X_1 + ... + X_n.

The i-th bid causes b* to be updated if and only if the highest bid among the first i bids comes at position i. Since all orders are equally likely, the probability that the highest bid among the first i bids comes at position i is 1/i. Thus, the expected value of X_i is E[X_i] = 1/i.

Now we can use the linearity of expectation to find the expected number of times b* is updated. The expected value of X is the sum of the expected values of X_i:

$$E[X] = E[X_1] + E[X_2] + ... + E[X_n] = 1/1 + 1/2 + ... + 1/n$$

This sum is the Harmonic series, and its sum H_n is approximately ln(n) + γ, where γ is the Euler-Mascheroni constant (approximately 0.5772).
Thus, the expected number of times that b* is updated is E[X] = H_n = Θ(log n).

Q12 (5 Points)

In a co-op, you develop an algorithm for a content delivery network like YouTube or Miley.com. Suppose that in a typical minute, you get a k (e.g. a bazillion) content requests, and each needs to be served from one of your *n* servers. Your algorithm is randomly assign each job to a random server.

A) What is the expected number of jobs per server?

Ans:

There are k jobs and n servers, so the expected number of jobs per server is k/n.

B) What is the probability that a server gets twice the average load? That is, 2 times the expected number of jobs? (A bound is acceptable)

Ans:

We can use the Chernoff bounds to find an upper bound on this probability.

Let X be the random variable representing the number of jobs assigned to a server.

The expected value of X is μ = k/n. We are looking for the probability that X >= (1+δ)μ, where δ = 1 (since we want twice the expected number of jobs).

Using the Chernoff bound formula for the probability above the mean:

$$Pr[X >= (1+\delta)\mu] <= (e^\delta / ((1+\delta)^(1+\delta)))^(\mu)$$

Plugging in the values for μ and δ:

$$Pr[X >= 2k/n] <= (e^1 / (2^2))^(k/n) = (e/4)^(k/n)$$

Since e/4 is less than 1, this probability will decrease exponentially.

C) What is the probability that a server gets no load? That is, no jobs? (A bound is acceptable)

Ans:

We can again use the Chernoff bounds to find an upper bound on this probability.

We are looking for the probability that X < (1-δ)μ, where δ = 1 (since we want no jobs assigned to a server)

Using the Chernoff bound formula for the probability below the mean:

$$Pr[X < (1-\delta)\mu] <= e^{((-\delta^2)*(\mu/2))}$$

Plugging in the values for μ and δ:

$$Pr[X < 0] <= e^{((-1^2)*(k/2n))} = e^{(-k/2n)}$$

An alternative way to calculate this probability is to use the fact that the probability that a server gets a job is 1/n, and the probability that a server doesn't get a job is 1-1/n. The probability that a server gets no jobs after k requests is thus (1-1/n)^k.

Q13 (5 Points)

Does the state-flipping algorithm always terminate? If so, why?

Ans:

The state-flipping algorithm is applied to Hopfield networks, which are a type of recurrent neural network that can store patterns and retrieve them when given partial information. In these networks, nodes can have either an "on" or "off" state. The state-flipping algorithm involves iteratively updating the state of unsatisfied nodes until the network reaches a stable configuration.

We want to prove that the state-flipping algorithm always terminates, which implies that there is always a stable configuration. To do this, we will introduce a measure of progress, Φ(S), that will help us understand the behavior of the algorithm.

Let Φ(S) be the total absolute weight of all good edges in the network:

$$\Phi(S) = \sum(\_good\ e)\ |W\_e| - (2)$$

Consider a node u in the network. If we flip its state, all good edges incident to u become bad, and all bad edges become good, while other edges remain unchanged. Let g_u and b_u denote the total absolute weight on good and bad edges incident to node u, respectively. After flipping the state of node u, we have:

$$\Phi(S') = \Phi(S) - g\_u + b\_u$$

Since node u was unsatisfied, b_u > g_u. As both g_u and b_u are integers, it follows that b_u > g_u + 1. Substituting this inequality into the equation for Φ(S'), we get:

$$\Phi(S') = \Phi(S) + 1$$

The maximum value of Φ(S') is W, which is the total weight of the network. As Φ(S') increases with each iteration, there must be a limit to the number of iterations in the state-flipping algorithm.

Thus, we can conclude that the state-flipping algorithm will always terminate, implying that every Hopfield network has a stable configuration. This result ensures that the Hopfield network can store and retrieve patterns, making it a useful model for associative memory and pattern recognition tasks.

Q14 (5 Points)

What is the probability of getting exactly no heads after flipping four coins?

Ans:

The probability of getting exactly no heads after flipping four coins can be calculated using the binomial distribution formula:

$$P(X=k) = (n \text{ choose } k) * p^k * (1-p)^{(n-k)}$$

where:

n is the number of trials (flips)
k is the number of successes (heads) we want to achieve
p is the probability of success in each trial
(n choose k) is the number of combinations of n items taken k at a time, denoted by C(n, k)

In this case, we have:

n = 4 (flipping four coins)
k = 0 (we want exactly no heads)
p = 1/2 (the probability of getting a head in each flip)

Applying these values to the formula:

$$\begin{aligned} P(X=0) &= C(4, 0) * (1/2)^0 * (1 - 1/2)^{(4-0)} \\ &= 1 * 1 * (1/2)^4 \\ &= 1/16 \end{aligned}$$

So the probability of getting exactly no heads after flipping four coins is 1/16.

Q15 (10 Points)

Suppose you are using a randomized version of quicksort on a very large set of real numbers, and you would like to pick a pivot by sampling. Suppose you sample a subset uniformly at random (with replacement) and use that to estimate the value of your pivot. Use a Chernoff-bound to calculate your confidence in your approximate pivot estimate. You can choose your confidence level (typical is 90%, 95% or 99% confidence) the distribution of the numbers and your sample size.

Ans:

Let's say we want to pick a pivot for a randomized version of quicksort using a sample size with a 95% confidence level and a confidence interval of +/- 5%.

We can use the Chernoff bound to calculate the required sample size. First, let's assume we have a uniform distribution of real numbers between 0 and 1.

Suppose we sample n numbers and compute their mean μ. We want to estimate the true mean μ_true within a range of +/- ε with a probability of at least 1 - δ. In this case, ε = 0.05 and δ = 0.05 (for a 95% confidence level).

Recall the Chernoff bound for values above and below the mean:

$$Pr[\mu \geq (1 + \varepsilon)\mu\_true] \leq e^{(-\varepsilon^2 * \mu\_true * n / 3)}$$

$$Pr[\mu \leq (1 - \varepsilon)\mu\_true] \leq e^{(-\varepsilon^2 * \mu\_true * n / 2)}$$

To bound the probability of our estimate being off by more than ε in either direction, we want:

$$e^{(-\varepsilon^2 * \mu\_true * n / 3)} \leq \delta/2$$
$$e^{(-\varepsilon^2 * \mu\_true * n / 2)} \leq \delta/2$$

We can use the fact that the expected value of a uniformly distributed random variable between 0 and 1 is 0.5 (μ_true = 0.5). Now, solving for n in the above inequalities:

$$n \geq 3 * \ln(2/\delta) / (\varepsilon^2 * \mu\_true) \text{ for values above the mean}$$
$$n \geq 2 * \ln(2/\delta) / (\varepsilon^2 * \mu\_true) \text{ for values below the mean}$$

Substituting the values ε = 0.05, δ = 0.05, and μ_true = 0.5, we get:

$$n \geq 3 * \ln(40) / (0.0025 * 0.5) \text{ for values above the mean}$$
$$n \geq 2 * \ln(40) / (0.0025 * 0.5) \text{ for values below the mean}$$

Calculating the sample size for both cases:

$$n \geq 1471.15 \text{ for values above the mean}$$
$$n \geq 981.44 \text{ for values below the mean}$$

To achieve a 95% confidence level with a confidence interval of +/- 5%, we should sample at least 1472 numbers, which is the largest value obtained from the above calculations using Chernoff bounds.

Q16 (10 Points)

Consider a server cluster with p processes and n servers (p > n). Your load balancing algorithm selects a server at random to place any new process, all equally likely.

A) What is the expected number of processes in each server?

Ans:

The expected number of processes in each server is obtained by dividing the total number of processes (p) by the total number of servers (n). This is because each server has an equal probability of receiving a process. So, the expected number of processes in each server is:

$$E[\text{processes per server}] = p/n$$

B) How big can the difference in server load be? Give an explicit probability.

Ans:

The largest difference in server load would be when one server gets all the processes, and the others get none. The probability of this happening is the probability of a server receiving all p processes:

$$P(\text{a server gets all processes}) = (1/n)^p$$

C) What is the likelihood that a server is k times an average server?

Ans:

To find the likelihood that a server has a load k times the average load, we use the Chernoff bound for values above the mean:

The average server load is $p/n = \mu$.

Let $1+\delta=k$. Then, $k-1=\delta$.

$Pr[X>(1+\delta)\mu] < [e^{\delta}/((1+\delta)^{(1+\delta)})]^{\mu}$

$Pr[X>(k*\mu)] < [e^{(k-1)}/((k)^{(k)})]^{(p/n)}$

D) What is the likelihood that a server is 1/k below an average server?

Ans:

To find the likelihood that a server has a load 1/k times below the average load, we use the Chernoff bound for values below the mean:

The average server load is $p/n = \mu$.

Let $1-\delta = 1/k$. Then, $1-1/k = \delta$.

$Pr[X<(1-\delta)\mu] <= e^{(-\delta^2\mu/2)}$

$Pr[X<(1-\delta)\mu] < e^{(-(1-1/k)^2p/2n)}$

This provides the likelihood that a server's load is 1/k times below the average server load. Keep in mind that these probabilities are bounds and not exact values, but they can give us an understanding of how the load may be distributed among the servers.