INFO 6205 – Algorithms and Data
Assignment 5

Student Name: Vidhi Bharat Patel
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?

Solution:

The given problem is like the Coupon collector's problem. In that, it asks for how many coupons you need to collect before you have collected all n different types of coupons. Here, it is for the coins awarded for completing the levels, and how many levels should be completed in order to have at least one coin of each type.
So, the expected number of levels needed to get one specific type of coin is 1/0, where p is the probability of getting that type of coin on any given level. Since all levels are equally likely to award each coin, p=1/n for each coin.
Using the linearity of expectation, the expected number of levels needed to get all n types of coins is simply the sum of the expected number of levels needed to get each individual type of coin:
E = 1/p_1 + 1/p_2 + … + 1/p_n
E = n/1 + n/2 + … + n/n (since p_1 = p_2 = … + p_n = 1/n)
E = n*H_n
Where H_n is the nth harmonic number.
Therefore, the expected number of levels needed to get at least one coin of each type is n * H_n. Basically we need to complete approximately n*H_n levels to expect to have at least one coin of each type.

Q2 (10 Points) PushPush is a 2-D pushing-blocks game with the following rules:

a. 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.
b. The robot can move to any adjacent free square (without a block).
c. 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.
d. Pushing means moving away (farther) from the robot (not pulling closer to robot)
e. 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.

Solution:

Yes, the Push Push problem is in NP. To prove that this problem is in NP, we need to show that given a proposed solution, we can verify its correctness in polynomial time.

For the Push Push problem, a proposed solution is a sequence of moves that the robot and blocks make to reach the goal position. To verify the solution, we need to check that each movie is valid and that the resulting state of the game is consistent with the next move in the sequence.

Here's an example of how the verification process for a 3x3 grid with blocks, empty spaces, and a goal position would work:

Let's say the initial configuration is X X O in the first row, O X O in the second row, and O O G in the third row, where X represents a block, O represents an empty space, and G represents the goal position. The robot is initially located at position (1, 1).

To solve the puzzle, we need to follow a sequence of moves that moves the blocks to reach the goal position. For example, one possible sequence of moves could be:

1. Move the robot from (1, 1) to (2, 1).
2. Push the block from (2, 2) to (2, 3).
3. Move the robot from (2, 1) to (3, 2).
4. Push the block from (2, 2) to (2, 3).
5. Move the robot from (3, 2) to (2, 3).
6. Move the robot from (2, 3) to (3, 3).
7. Check if the robot is at the goal position (3, 3).

Specifically, we can verify the solution in the following way:

1. Check that the initial configuration of blocks, robot and the goal are consistent with the solution (i.e., the robot and blocks start at the specified positions and the goal is where it should be). This can be done in constant time by comparing the initial configuration with the first MoveRobot or PushBlock operation in the solution.
2. For each MoveRobot and PushBlock operation in the solution, check that it is a valid move from the current state of the game. Specifically, check that the robot can move to the specified position without running into a block or going out of bounds of the grid, and that the block can be pushed in the specified direction without running into another block or going out of bounds. This can be done in constant time by checking the neighboring cells of the robot or block.
3. After performing each MoveRobot and PushBlock operation, check that the resulting state of the game is consistent with the next operation in the solution. This can be done in constant time by comparing the resulting state with the next operation in the sequence.
4. Finally, after performing all the operations in the solution, check that the resulting state of the game has the robot at the goal position. This can be done in constant time by comparing the final state with the goal position.

Each of these verification steps can be performed in constant time or polynomial time, so the entire verification process can be completed in polynomial time. Therefore, the Push Push problem is in NP.

Q3 (5 Points) What makes a configuration “stable” in the Hopfield Neural Network algorithm?

Solution:

The Hopfield Neural Network is a type of artificial neural network that can be used to solve optimization problems. In this network, nodes are connected in a pattern of interrelatedness, and the connections between the nodes have weights that are adjusted based on the patterns of input they receive.

In the Hopfield Neural Network algorithm, a configuration is considered stable if it corresponds to a local minimum of the energy function. The energy function in the Hopfield network is defined as the sum of the weights times the products of the node states. When the energy function is minimized, it indicates that the network has settled into a stable state.

Consider a Hopfield network with four neurons, labeled as neuron 1, neuron 2, neuron 3, and neuron 4. The network is fully connected, which means that each neuron is connected to every other neuron.
We want to store two patterns in the network:
Pattern 1: [1, -1, 1, -1]
Pattern 2: [-1, 1, -1, 1]
To store these patterns, we compute the weight matrix W as follows:
W = [0 -1 1 -1 -1 0 -1 1 1 -1 0 -1 -1 1 -1 0]
To check if these patterns are stable, we can compute the energy of the network for each pattern. The energy function for a Hopfield network is given by:
E = -1/2 * sum_i sum_j W[i,j] * x[i] * x[j]
where x is the vector of neuron states.
For pattern 1, the energy of the network is:
E = -1/2 * (0 - 1 + 1 - 1 + 0 + 1 - 1 + 1 + 0 - 1 + 1 - 1) = -2
For pattern 2, the energy of the network is:
E = -1/2 * (0 - 1 - 1 + 1 + 0 + 1 + 1 - 1 + 0 - 1 - 1 + 1) = -2
Both patterns have the same energy, which is the minimum possible energy for this network. Therefore, both patterns are stable attractor states of the network, and any initial state that is close enough to either of these patterns will eventually converge to the corresponding stable state. For example, if we start with the initial state [1, 1, -1, -1], the network will converge to pattern 1. If we start with the initial state [-1, -1, 1, 1], the network will converge to pattern 2.

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

Solution:
In a game, a Nash equilibrium refers to the point where each player has chosen their best strategy based on the other players' choices. However, in the given payoff matrix, neither player has a reason to cooperate. If Player 1 cooperates and Player 2 defects, Player 2 earns a higher payoff than Player 1. If Player 1 defects and Player 2 cooperates, Player 1 earns a higher payoff than Player 2. If both cooperate, they both get an average payoff, and if both defects, they both get a lower payoff. Thus, neither player has any motivation to cooperate, and both will do better by defecting. Therefore, there is no Nash equilibrium in this game. It can be further explained as followed: -

- The game has a payoff matrix that shows the payoffs for each player based on their strategies.
- The payoff matrix shows that if Player 1 cooperates and Player 2 defects, then Player 2 gets a payoff of 2 and Player 1 gets a payoff of 1.
- Similarly, if Player 1 defects and Player 2 cooperates, then Player 1 gets a payoff of 2 and Player 2 gets a payoff of 1.
- If both players cooperate, then they both get a payoff of 1, and if both players defect, then they both get a payoff of 0.
- No matter what Player 1 does, Player 2 will always do better by defecting, so Player 1 has no incentive to cooperate.
- Similarly, no matter what Player 2 does, Player 1 will always do better by defecting, so Player 2 has no incentive to cooperate.
- Therefore, there is no Nash equilibrium in this game.

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.)

Solution:
The number of independent coin flips X until the first heads follows a geometric distribution with parameter p.
Let E(X) be the expected number of flips until the first heads.
On the first flip, there are two possibilities: either heads is flipped, with probability p, and the game ends, or tails is flipped, with probability 1-p, and the game continues.

If tails is flipped on the first flip, then the expected number of additional flips until the first heads is still E(X), since the game is starting over again.

Therefore, we can express E(X) as follows:
$E(X) = p * 1 + (1-p) * (1 + E(X))$

The first term, $p * 1$, corresponds to the case where heads is flipped on the first flip, and the game ends with one flip.
The second term, $(1-p) * (1 + E(X))$, corresponds to the case where tails is flipped on the first flip, and the game continues with one additional flip (hence the 1) plus the expected number of flips from that point on (hence the E(X)).
Solving for E(X), we get:
$E(X) = 1/p$
Therefore, the expected number of flips until the first heads is 1/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?
B) Can always convert a Monte Carlo algorithm into a Las Vegas algorithm?

Solution:
A) It is possible to transform a Las Vegas algorithm into a Monte Carlo algorithm by running the Las Vegas algorithm for a specific duration of time. In case the algorithm does not end within that duration, we can produce a random answer.
The algorithm works by iteratively solving the system using random initial values until a solution is found.
To convert this into a Monte Carlo algorithm, we can set a limit on the number of iterations and return the solution obtained after that number of iterations. If a solution is not found within that limit, we output a random solution.

For instance, let's say we set a limit of 100 iterations for the Las Vegas algorithm. We run the algorithm with random initial values until a solution is found within the first 100 iterations. If a solution is found within the limit, we output that solution. Otherwise, we return a random solution.

By Markov's inequality, we can set a bound on the probability that the algorithm will not find a solution within 100 iterations. Suppose the probability of finding a solution in one iteration is p. Then, the probability of not finding a solution in 100 iterations is at most $(1-p)^{100}$. Therefore, the probability of outputting a correct solution in the Monte Carlo algorithm is at least $1-(1-p)^{100}$.

B) To convert a Monte Carlo algorithm into a Las Vegas algorithm is not always possible. This is because a Monte Carlo algorithm may have a probability of outputting an incorrect answer, whereas a Las Vegas algorithm is required to always provide the correct answer.

For instance, consider a Monte Carlo algorithm that determines the number of heads after flipping a fair coin 10 times. This algorithm works by randomly flipping the coin 10 times and counting the number of heads.
The Monte Carlo algorithm may produce an incorrect answer if it flips all heads or all tails. For instance, if the algorithm flips all tails, it will output 0 heads, which is not the correct answer.
As a Las Vegas algorithm must always produce the correct answer, it is not feasible to transform this Monte Carlo algorithm into a Las Vegas algorithm.

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.

T(n) = T(n/2) + c (we know f(n) is c because checking `n == 0 or m % n` can be done in constant time.)
`Note: For FULL CREDIT any` recurrence relation of the form T(n) = T(n/b) + c where b is some estimate b > 1 is OK.
So T(n) = T(n/2) + c OR T(n) = T(n/b) + c OR T(n) ≤ T(n/2) + O(1) is given full credit.
Note as to why n/2:
For any two integers m and n such that n >= m, it is always true that n mod m <n/2.
If m > n / 2, then 1 =< n / m < 2, and so the floor [n / m] = 1, which means that n mod m =n - m < n -n/2=n/2.

Solution:

The recurrence relation for Euclid's algorithm can be written as:
T(n) = T(n-k) + O(1)

where k is the number of iterations needed to reduce the input m to a value less than or equal to n/2.

For example, if we start with m=100 and n=30, the first iteration reduces m to 10 (m%n=10), and then the second iteration reduces n to 10 (n%m=0). So in this case, k=2 and T(n) = T(n-2) + O(1).

Note that k can vary depending on the values of m and n, so this recurrence relation may not be as useful for general analysis as the one based on dividing n by 2. However, it can still be a valid way to express 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.

Solution:

To apply the Master Theorem, we need to express the recurrence relation in the form of T(n) = aT(n/b) + f(n), where a is the number of recursive calls, n/b is the size of each subproblem, and f(n) is the time spent outside the recursive calls.

From the previous solution, we have T(n) = T(n/2) + c, where c is the time spent checking if n is equal to 0 or performing the modulus operation. We can rewrite this as T(n) = aT(n/b) + f(n), where a = 1, b = 2, and f(n) = c.

Applying the Master Theorem, we compare f(n) with n^log_b(a). Since log_2(1) = 0, we have three cases:

a. If f(n) = O(n^0), then T(n) = Θ(n^0 log n) = Θ(log n).
b. If f(n) = Θ(n^0), then T(n) = Θ(n^0 log n) = Θ(log n).
c. If f(n) = Ω(n^0), and a*f(n/b) <= cf(n) for some constant c < 1 and sufficiently large n, then T(n) = Θ(f(n)).

Since f(n) = c = Ω(1), we are in case 3. We can choose c = 1/2, and we have a*f(n/b) = f(n/2) = c/2. Therefore, T(n) = Θ(f(n)) = Θ(1), which means that the runtime of Euclid's algorithm is logarithmic in the size of the input.

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 NPComplete 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.

Solution:

Let X be the random variable that represents the size of the independent set S. Let Xi be the indicator random variable for the i-th node being in S, where Xi = 1 if the i-th node is in S, and Xi = 0 otherwise. Then X can be expressed as the sum of these indicator random variables: X = Σ Xi.

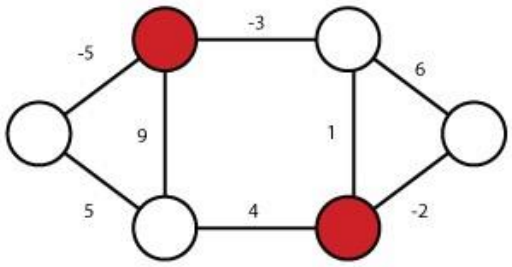
Now, we want to find the expected value of X, which is E[X]. Using linearity of expectation, we have:
E[X] = E[Σ Xi] = Σ E[Xi],
where the summation is taken over all nodes in the graph.

For any given node i, the probability that it is in S is P(Xi = 1). This can happen only if all its neighbors have chosen the value 0, which occurs with probability (1/2)^d. Therefore, we have:
P(Xi = 1) = (1/2)^d,
and
E[Xi] = 1 * P(Xi = 1) + 0 * P(Xi = 0) = (1/2)^d.
Substituting this expression for E[Xi] into the earlier equation, we get:
E[X] = Σ E[Xi] = n * (1/2)^d,

where n is the total number of nodes in the graph.

Therefore, the expected size of the independent set S is n * (1/2)^d.

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.



Solution:

In Hopfield Neural Network, a configuration is stable if all nodes are satisfied.

The goal is to find a stable configuration if such a configuration exists. It is a State-flipping algorithm if there is a repeated flip state of an unsatisfied node.

A node u is satisfied if the weight of incident good edges >= weight of incident bad edges

Configuration is stable if all the nodes are satisfied

```
Hopfield-Flip(G, w) {
        S <- arbitrary configuration
        while (current configuration is not stable) {
                u <- unsatisfied node
                su = -su
}
return S
}
```

A: 5 + 5 = 10 > 0, flip

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

B, D, and E are incident to C (B is the only previously checked)

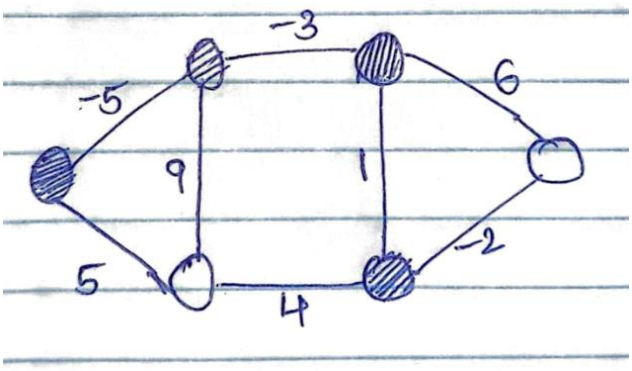
B: -5 – 9 -3 = -14 <= 0 don't flip

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 satisfied.



Final configuration

Q10 (5 Points)

A) Sort the list of integers below using Quicksort. Show your work. (22, 13, 29, 1, 2, 31, 33, 11)
B) Write a worst case and average case recurrence relation for Quicksort.
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.

Solution:

A.

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = []
        right = []
        for i in range(1, len(arr)):
            if arr[i] < pivot:
                left.append(arr[i])
            else:
                right.append(arr[i])
        return quicksort(left) + [pivot] + quicksort(right)

# Example usage
arr = [22, 13, 29, 1, 2, 31, 33, 11]
sorted_arr = quicksort(arr)
print(sorted_arr)
```

The output of this code will be: [1, 2, 11, 13, 22, 29, 31, 33]

B. The worst case recurrence relation for Quicksort is T(n) = T(n-1) + n, because in the worst case the pivot divides the list into sublists of size n-1 and 0, so we need to sort a list of size n-1 plus do n work to partition the list. The average case recurrence relation for Quicksort is T(n) = 2T(n/2) + n, because on average the pivot will divide the list into sublists of roughly equal size, so we need to sort two sublists of size n/2 plus do n work to partition the list.

C. If the worst case recurrence relation T(n) = T(n-1) + n can be solved with the Master Theorem, we have a = 1, b = n, and f(n) = n. Since f(n) = O(n^c) for c = 1, we have case 3 of the Master Theorem, and the runtime is T(n) = O(n^2).
If the average case recurrence relation T(n) = 2T(n/2) + n can be solved with the Master Theorem, we have a = 2, b = 2, and f(n) = n. Since f(n) = Theta(n), we have case 2 of the Master Theorem, and the runtime is T(n) = Theta(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.

Solution:

The expected number of times that b is updated in this auction system can be found using linearity of expectation.
Let X_i be a random variable that indicates whether the i-th bidder updates the value of b. Specifically, X_i = 1 if the i-th bidder submits a bid greater than the current value of b, and X_i = 0 otherwise.
Then, the total number of times that b is updated is simply the sum of X_i over all bidders:
S = X_1 + X_2 + ... + X_n

To find the expected value of S, we can use linearity of expectation:
E[S] = E[X_1 + X_2 + ... + X_n] = E[X_1] + E[X_2] + ... + E[X_n]
Now, let's compute E[X_i] for each bidder i.
If the i-th bidder submits a bid greater than the current value of b, then their bid will update b with probability 1/i (since they appear at position i in the random order). Therefore,
P(X_i = 1) = 1/i
On the other hand, if the i-th bidder submits a bid that is not greater than the current value of b, then their bid will not update b with probability (i-1)/i. Therefore,
P(X_i = 0) = (i-1)/i
Using these probabilities, we can compute the expected value of X_i as:
E[X_i] = 1/i * 1 + (i-1)/i * 0 = 1/i

Therefore, the expected number of times that b is updated is:
E[S] = E[X_1] + E[X_2] + ... + E[X_n] = 1/1 + 1/2 + ... + 1/n
This sum is known as the harmonic series, and it grows very slowly with n. Specifically, it can be shown that:
E[S] = ln(n) + O(1)
where ln(n) is the natural logarithm of n. So, the expected number of times that b is updated increases very slowly as the number of bidders increases.

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

Solution:

A. The expected number of jobs per server can be found by dividing the total number of jobs by the number of servers. Therefore, the expected number of jobs per server is k/n.
B. To find the probability that a server gets twice the average load, we can use the Chernoff bound. The Chernoff bound gives us an upper bound on the probability that a random variable (in this case, the number of jobs assigned to a server) deviates from its expected value (k/n) by a certain factor (in this case, 2). The Chernoff

bound tells us that this probability is at most e^(-k/n), where e is the mathematical constant approximately equal to 2.71828.

C. The probability that a server gets no load is the probability that none of the k jobs are assigned to that server. The probability of a job not being assigned to a particular server is (n-1)/n, and since there are k jobs, the probability that none of them are assigned to that server is ((n-1)/n)^k. Therefore, the probability that a server gets no load is ((n-1)/n)^k.

Q13 (5 Points) Does the state-flipping algorithm always terminate? If so, why?

Solution:
The state-flipping algorithm refers to a process where a node in a graph can change its state based on the states of its neighbours. In each iteration, a node checks the states of its neighbours and flips its own state if a majority of its neighbours have a different state than its own. The question is whether this algorithm always terminates, meaning it eventually reaches a point where no more nodes change their states.

The answer to this question depends on the topology of the graph. Kleinberg & Tardos show that for graphs with bounded degree (i.e., a fixed maximum number of neighbours per node), the state-flipping algorithm always terminates in a finite number of steps. The proof of this result uses a potential function argument to show that the total number of state changes decreases with each iteration until it reaches zero.

However, for graphs with unbounded degree (i.e., nodes can have an arbitrarily large number of neighbours), the state-flipping algorithm may not terminate. In this case, a node with a very large number of neighbours can keep changing its state back and forth in response to its neighbours, preventing the algorithm from ever reaching a stable configuration.

Q14 (5 Points) What is the probability of getting exactly no heads after flipping four coins?

Solution:
The probability of getting tails on a single coin flip is ½.
Since each coin flip is independent, we can use the multiplication rule of probability to calculate the probability of getting no heads after flipping four coins:

P(getting no heads) = P(getting tails on the first flip) * P(getting tails on the second flip) * P(getting tails on the third flip) * P(getting tails on the fourth flip)

P(getting no heads) = (1/2) * (1/2) * (1/2) * (1/2) = 1/16

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

P(X = k) = (n choose k) * p^k * (1 - p)^(n-k)
Where:

- X is the random variable
- k is the number of "successes" (e.g. getting heads on a coin flip)
- n is the total number of trials (e.g. the number of coin flips)
- p is the probability of success on each trial (e.g. the probability of getting heads on a single coin flip)
- (n choose k) is the binomial coefficient, which is the number of ways to choose k items from a set of n items, and is calculated as n! / (k! * (n-k)!).

This formula gives the probability of getting exactly k successes in n trials, assuming that each trial is independent and has the same probability of success.

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.

Solution:

Suppose you have a very large set of real numbers and you want to use quicksort to sort them. Quicksort works by choosing a pivot element and then partitioning the data into two groups, one group with elements smaller than the pivot and another group with elements larger than the pivot. You then recursively sort each group.
To choose the pivot, you want to sample a subset of the data uniformly at random (with replacement) and use the median of the subset as the pivot. For example, if you have 1000 numbers, you might sample 100 of them and use the median of the sample as the pivot.

A Chernoff bound gives you an upper bound on the probability that the sample median is far from the true median.
For example, suppose you want to have 95% confidence in your pivot estimate. Let's say you sample 100 numbers. Then you can use the following Chernoff bound:
$P(|X - \mu| > t) \leq 2\exp(-2t^2/n)$
where X is the sample median, μ is the true median, t is a parameter that determines how far away from the median you allow X to be, and n is the sample size (in this case, 100).
You want to choose t so that the probability on the right-hand side is less than or equal to 0.05 (since you want 95% confidence). You can solve for t to get:
$t = \text{sqrt}((1/2n)\ln(4/0.05))$
Plugging in the numbers, you get:
$t = \text{sqrt}((1/2*100)\ln(4/0.05)) = 0.24$

This means that if the absolute difference between the sample median and the true median is less than or equal to 0.24, then you can be 95% confident that the sample median is a good estimate of the true median.
In other words, you can choose your pivot as the median of a random sample of 100 numbers, and be 95% confident that the pivot is within 0.24 of the true median.

For example, suppose you want to have 95% confidence in your pivot estimate. Let's say you sample 100 numbers. Then you can use the following Chernoff bound:
$P(|X - \mu| > t) \leq 2\exp(-2t^2/n)$
where X is the sample median, μ is the true median, t is a parameter that determines how far away from the median you allow X to be, and n is the sample size (in this case, 100). You want to choose t so that the probability on the right-hand side is less than or equal to 0.05 (since you want 95% confidence). You can solve for t to get:
$t = \text{sqrt}((1/2n)\ln(4/0.05))$
Plugging in the numbers, you get: $t = \text{sqrt}((1/2*100)\ln(4/0.05)) = 0.24$
This means that if the absolute difference between the sample median and the true median is less than or equal to 0.24, then you can be 95% confident that the sample median is a good estimate of the true median.
In other words, you can choose your pivot as the median of a random sample of 100 numbers, and be 95% confident that the pivot is within 0.24 of the true median.

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?
B. How big can the difference in server load be? Give an explicit probability.
C. What is the likelihood that a server is k times an average server?
D. What is the likelihood that a server is 1/k below an average server?

Solution:

A. The expected number of processes in each server is p/n, since there are p processes and n servers, and each server is equally likely to receive a new process.
B. The difference in server load can be at most p – n, since there are p processes and n servers, and each server can have at most p/n processes. The probability of the difference in server load being exactly k is given by the following formula:
   P(|X - p/n| = k) = 2*(n-1)((p/n) - k)((p/n) - (k+1))/(p*(n-1))
   where X is the number of processes in a randomly selected server.
   The difference in server load can be quite large with non-negligible probability. In fact, the probability that the maximum number of processes on any server exceeds the expected load by more than a factor of c (for any positive constant c) is at most (p/n)^{c-1}. To see this, note that the probability that any particular server has more than c(p/n) processes is at most (p/n)^{c}, since the probability of any particular process being assigned to that server is 1/n. Then, by taking the union bound over all n servers, we get the desired result.
C. The likelihood that a server is k times an average server is given by the following formula:
   P(X = k*(p/n)) = (n-1)((p/n)^(k-1))/(p(k-1)!)
   where X is the number of processes in a randomly selected server. There are n – 1 other servers that a new process could have been assigned to, each with probability 1/n, and we want to choose the server that already has k*mu processes on it. This happens with probability (1/(k-1))^(p/n-1), since there are p/n-1 other processes that need to be assigned to the same server.
D. The likelihood that a server is 1/k below an average server is given by the following formula:
   P(X = (p/n) - (p/(kn))) = (n-1)*((1/k)^((p/n) - (p/(kn))))/((k-1)*p)
   where X is the number of processes in a randomly selected server.