Course: Randomized Algorithms by Dr. Zarei

Homework: HW1

Name: Mohammad Mohammadi

Student ID: 402208592

Question 2.3.36 [DARA_Hor]

Let us modify the algorithm RSAM as follows:

Input: A formula $\Phi=F1\wedge F2\wedge\ldots\wedge Fm$ over $\{x1,x2,\ldots,xn\}$ in CNF.

Step 1: Choose uniformly an assignment $(\alpha1,\alpha2,\ldots,\alpha n)$ to $x1,x2,\ldots,xn$ at random.

Step 2: Compute the number $r(\alpha1,\alpha2,\ldots,\alpha n)$ of clauses that are satisfied by $(\alpha1,\alpha2,\ldots,\alpha n)$.

Step 3: If $r(\alpha1,\alpha2,\ldots,\alpha n)\geq m/2$, then output $(\alpha1,\alpha2,\ldots,\alpha n)$, else repeat Step 1.

If this algorithm halts, then we have the assurance that it outputs an assignment satisfying at least half the clauses. Estimate the expected value of the number of executions of Step 1 of the algorithm (i.e., the expected running time of the algorithm).

Answer 2.3.36

To estimate the expected value of the number of executions of Step 1, we need to determine the probability that a random assignment satisfies at least m/2 clauses of the given formula Φ in Conjunctive Normal Form (CNF).

Let's denote:

P(success) as the probability that a random assignment satisfies at least m/2 clauses.

E(executions) as the expected number of executions of Step 1.

The algorithm repeats Step 1 until a successful assignment is found, which can be modeled as a geometric distribution because each trial (execution of Step 1) is independent, and the probability of success is the same for each trial. The expected value (mean) of the number of trials until the first success for a geometric distribution is given by 1/P(success).

So, to find E(executions), we need to calculate P(success).

However, calculating P(success) exactly for any formula Φ can be very complex because it depends on the specific structure of Φ. But we can provide some general insights.

Each clause in a CNF formula is a disjunction of literals (variables or their negations). The probability that a random assignment satisfies a single clause depends on the number of literals in the clause and their arrangement. However, without loss of generality, for a large enough formula with sufficiently mixed clauses, we can make a simplifying assumption to estimate P(success).

Assuming clauses are sufficiently varied and not all trivially satisfiable or unsatisfiable, we might approximate the probability of satisfying at least one literal in a clause as being better than random chance. However, the exact probability P(success) that a random assignment satisfies at least half of the clauses without specific details about the clauses is challenging to determine accurately.

For a very rough and optimistic approximation, if we consider that each clause has a probability greater than 0.5 (say p) to be satisfied by a random assignment due to the variety and number of literals, then the probability of satisfying at least m/2 out of m clauses could be assumed to be p^m for simplification, which is not accurate for calculating exact expectations but gives a sense that the probability decreases exponentially with the number of clauses m.

A more accurate approach would involve analyzing the distribution of clause satisfiability across all possible assignments, which can be highly variable depending on the structure of Φ. Without specific details about the distribution and types of clauses in Φ, we cannot accurately calculate P(success).

But to calculate the expected value precisely under significant simplifications and assumptions about the nature of the CNF formula Φ, we will consider a simplified scenario. Let's make the assumption that each clause in Φ is independent of the others and that the probability of a random assignment satisfying any single clause is p. This simplification allows us to treat the satisfaction of each clause as a Bernoulli trial with success probability p.

Given:

- A formula Φ in CNF with m clauses.
- A random assignment satisfies a single clause with probability p.
- We consider the clauses to be independent of each other.

To find the expected number of executions of Step 1 until an assignment satisfies at least half of the clauses, we need to determine the probability P(success) that a random assignment satisfies at least m/2 of the clauses.

Let's consider the probability P(success) as the probability of satisfying at least m/2 clauses. Since each clause is satisfied with probability p independently, the distribution of the number of satisfied clauses out of m follows a binomial distribution B(m,p).

The probability of satisfying exactly k clauses is given by: $P(X = k) = \binom{m}{k} p^k (1-p)^{m-k}$

Hence, the probability of satisfying at least m/2 clauses is:
$P(\text{success}) = P(X \geq m/2) = \sum_{k=\lceil m/2 \rceil}^{m} \binom{m}{k} p^k (1-p)^{m-k}$

For a geometric distribution, where each trial is independent, and the probability of success is P(success), the expected number of trials E(executions) until the first success is: $E(\text{executions}) = \frac{1}{P(\text{success})}$

Given the complexity of directly computing P(success) for general m and p, we'll approximate it using the formula for the probability of satisfying at least half the clauses. Let's calculate E(executions) for a simplified case where p is assumed to be 0.5, implying a random chance of satisfying each clause, which is a common assumption for simplification.

We'll compute the expected number of executions under these simplifications. Let's calculate it for a given m with p=0.5.

Under the simplification that each clause in the CNF formula is independent and has a random chance (0.5) of being satisfied by a random assignment, for a formula with 10 clauses (m=10), the expected number of executions of Step 1 until an assignment satisfies at least half of the clauses is approximately 1.605. This means, on average, about 1.605 attempts are expected before finding an assignment that satisfies at least half of the clauses under these assumptions.

Question 2.4.50 [DARA_Hor]

Consider the following communication task for the computers RI and RII. RI has 2n bits, x1,x2, ...,x2n, and RII has an integer j $\in$ {1, ...,2n}. RI does not know any bit of the input of RII, and RII does not have any information about the input of RI. We allow the submission of one message from RI to RII only, and after that we require that RII provides the bit xj as its output. The complexity of such a protocol is the length of the only binary message communicated.

Part 1. Prove that every deterministic protocol for this task must allow messages of length 2n.

Part 2. Does there exist a Las Vegas protocol that is allowed to output “?” and that solves this task within complexity n + 1?

Answer 2.4.50

Part 1: Deterministic Protocol Complexity

To understand why every deterministic protocol for this task must allow messages of length 2n, consider the requirements of the task. RI has a sequence of 2n bits, and RII needs to identify the value of the bit at position j, where j is known only to RII.

In a deterministic protocol, the message sent from RI to RII must contain enough information for RII to determine the value of xj without any chance of error. Since RII's input j can be any number from 1 to 2n, and RII has no prior knowledge of RI's input bits, the only way for RI to ensure that RII can accurately determine xj for any j is to send all 2n bits. If the message were shorter than 2n bits, there would be at least one j for which RII could not determine xj with certainty, as the message would not contain enough information to distinguish between all possible inputs.

Therefore, the length of the message must be at least 2n bits to ensure that RII can always correctly output xj. This proves that every deterministic protocol for this task must allow messages of length 2n.

Part 2: Las Vegas Protocol Complexity

A Las Vegas protocol is a randomized algorithm that always produces either a correct result or a special symbol indicating failure (in this case, "?") and never produces an incorrect result. The question is whether there exists a Las Vegas protocol for this task with complexity n + 1, meaning the length of the message from RI to RII is n + 1 bits.

To achieve this, we can use a randomization strategy. However, it's crucial to analyze whether such a strategy can guarantee the required correctness while limiting the message length to n + 1 bits.

Given that RII only needs to output a specific bit xj or "?", a potential strategy involves RI encoding its 2n bits into a shorter message using some form of randomized encoding. However, the challenge lies in ensuring that this encoding allows RII to always correctly determine xj or know when it cannot (and hence output "?") with just n + 1 bits of information.

This task, at first glance, seems to defy direct reduction to a smaller message size while maintaining the Las Vegas property, because RII needs to correctly guess a specific bit from 2n options based only on n + 1 bits of information from RI, without any error. Given the nature of the Las Vegas protocol, which cannot err when it decides to output a bit value, and considering the information theoretical limit that n + 1 bits cannot uniquely represent 2n bits without some form of prior agreement or additional information, it appears challenging to design such a protocol directly.

For a Las Vegas protocol to exist with complexity n + 1, it would have to ingeniously encode the 2n bits into n + 1 bits in such a way that RII can with certainty either identify xj or acknowledge the impossibility to do so (and output "?"). The critical limitation here is the need for the protocol to never err while also dealing with the exponential growth of possible inputs as n increases.

Considering these factors, the key lies in the ability to use randomness to somehow compress the necessary information into n + 1 bits without losing the certainty required for a Las Vegas protocol. This is a highly non-trivial task and, without a specific encoding/decoding scheme that leverages properties of randomness to ensure correctness, it seems improbable. The challenge primarily lies in compressing the information while maintaining the ability to accurately decode a specific bit or rightfully claim uncertainty.

In essence, the existence of such a Las Vegas protocol would require a novel approach to information encoding and randomness that overcomes the apparent limitations imposed by the task's requirements and the information theoretical constraints. Without an explicit scheme that demonstrates how this could be achieved, we cannot conclusively say such a protocol exists under the standard interpretations and limitations of information theory and randomness in communication protocols.

But to analyze the existence of a Las Vegas protocol with complexity n+1 for the communication task described, we can make some assumptions and explore possible strategies that leverage the properties of randomness to encode and decode information efficiently.

Assumptions

- **Randomized Strategy:** RI can use a randomized strategy to encode its 2n bits into a shorter message. This strategy involves using randomness to decide how to encode the bits, which could potentially allow RII to decode a specific bit xj or acknowledge when it cannot determine xj and output "?".
- **Error Detection Capability:** The encoding scheme allows RII to detect with certainty when it cannot reliably determine xj, thus maintaining the Las Vegas protocol's requirement of never outputting an incorrect result.

Potential Strategy One potential strategy involves using a hash function that maps the 2n bits into n bits, along with an additional bit to indicate the certainty or uncertainty of decoding. The key challenge is designing this scheme so that it meets the Las Vegas protocol requirements.

- **Step 1: Hash Function:** RI uses a hash function H that takes the 2n bits as input and produces an n-bit hash. The choice of H is crucial; it must distribute its outputs evenly across the n-bit space to minimize collisions. A collision occurs when two different inputs to H (two different positions j and j') could result in the same hash output, potentially confusing RII.
- **Step 2: Encoding Certainty:** Along with the n-bit hash, RI sends an additional bit indicating whether RII can be certain of the output. This bit could be determined based on some property of the input bits or the hash output, such as whether the hash function output falls into a range of values known to encode a specific bit reliably.
- **Step 3: Decoding by RII:** Upon receiving the n+1-bit message, RII uses the hash and the additional bit to try to decode xj. If the additional bit indicates certainty, RII proceeds to extract xj based on the hash; if it indicates uncertainty, RII outputs "?".

Analysis

This strategy relies heavily on the properties of the hash function H and the mechanism for determining the additional certainty bit. For it to work, H must have certain properties:

- **Low Collision Rate:** The hash function must minimize the chances of two different positions j and j' having the same hash value, to avoid ambiguity in decoding.

- **Efficient Decoding:** There must be a straightforward way for RII to decode xj from the hash, assuming no collision and certainty indicated.

However, the fundamental challenge here is that any hash function mapping 2n bits to n bits inherently involves some loss of information, leading to potential collisions. The additional bit of information helps mitigate this by signaling when the hash might not be reliable, but this doesn't eliminate the fundamental issue of collisions.

Given these considerations, while a Las Vegas protocol with complexity n+1 could theoretically exist under specific assumptions about the hash function and encoding mechanism, designing such a protocol that meets all requirements, especially never incorrectly decoding xj, is highly challenging. The protocol's effectiveness would depend on the specific properties of the hash function and the additional certainty mechanism, which must be carefully designed to minimize collisions and ensure that RII can always either correctly decode xj or know when to output "?".

Question 2.4.53 [DARA_Hor]

Modify the 2MC(monte carlo) algorithm At to a randomized algorithm At' in such a way that At' takes the most frequent result as the output. What is the error probability of this modified algorithm At'?

Answer 2.4.53

To modify the 2MC (Monte Carlo) algorithm At to a randomized algorithm At' that takes the most frequent result as the output, we can follow a simple modification strategy. In the original 2MC algorithm, let's say At runs a certain computation twice and may output different results due to its probabilistic nature. The modification for At' involves running the algorithm multiple times instead of just twice, and then selecting the most frequently occurring result as the final output.

Modification Steps:

1. **Run the Algorithm Multiple Times:** Execute At a predefined number of times, say k, where k > 2 and ideally is an odd number to avoid ties.
2. **Collect Results:** Keep track of all the results from these k executions.
3. **Select Most Frequent Result:** Determine which result occurs most frequently among the k executions and use this as the final output of At'.

Error Probability of At':

Assuming At has a certain probability of error p (i.e., it produces the correct result with probability 1-p), the error probability of At' depends on the distribution of correct and incorrect results across the k executions. The modified algorithm At' reduces the overall error probability by relying on the majority rule, assuming that the probability of At producing the correct result is greater than 0.5.

To calculate the error probability of At', we would consider the probability that more than half of the k executions give the wrong result. This can be computed using the binomial distribution formula, considering the "success" in this context as receiving the incorrect result (to calculate the probability of error).

Let Pe be the error probability of At', p the error probability of a single execution of At, and k the total number of executions. Then, Pe can be calculated as the sum of the probabilities of more than half of the k executions being incorrect: $P_e = \sum_{i=\lceil \frac{k}{2} \rceil}^{k} \binom{k}{i} p^i (1-p)^{k-i}$

Where:

- i is the number of executions that produced an incorrect result.
- $\binom{k}{i}$ represents the binomial coefficient, calculating the number of ways to choose i incorrect results out of k executions.
- p^i is the probability of getting i incorrect results.
- $(1 - p)^{k-i}$ is the probability of the remaining executions producing the correct result.

Given that p < 0.5 (assuming the algorithm is more likely to be correct than not), running At multiple times and selecting the most frequent result significantly reduces the error probability compared to a single execution or even the original 2MC method. The exact value of Pe depends on the specific values of p and k. Choosing a larger k (particularly an odd number to avoid ties) further decreases Pe, enhancing the reliability of At'.

Question 2.4.56 [DARA_Hor]

Let A be a randomized algorithm computing a function F with Prob(A(x) = F(x)) ≥ 1/3 for every argument x of F. Assume that one is aware of the fact that Prob(A(x) = α) ≤ 1/4 for every wrong result α (i.e., that the probability of computing any specific wrong result is at most 1/4). Can this knowledge be used to design a useful randomized algorithm for F?

Answer 2.4.56

Given the information about the randomized algorithm A and its properties, we can indeed use this knowledge to design a more useful randomized algorithm for F that has a higher probability of producing the correct result. The key idea is to run A multiple times and use a voting or majority rule to decide on the final output. This approach leverages the fact that while A might produce incorrect results, the probability of it producing any specific incorrect result is relatively low (≤1/4).

Designing the Improved Algorithm:

1. **Multiple Executions:** Run the algorithm A a certain number of times, say n, where n is chosen based on the desired confidence level in the correctness of the final output.
2. **Collect and Analyze Results:** Keep track of all results from these n executions.
3. **Decision Rule:** Select the result that appears most frequently as the output of the improved algorithm. If there's a tie, or if the most frequent result does not appear significantly more often than others, the algorithm can output a special symbol (e.g., "?") to indicate uncertainty.

Why This Works:

- **High Probability of Correctness:** Given that Prob(A(x)=F(x))≥1/3, there's a reasonable chance that in multiple runs, the correct result will appear more frequently than any single incorrect result, especially since any specific incorrect result has a probability of ≤1/4.
- **Mitigating Specific Incorrect Outputs:** Since the probability of producing any specific wrong result is at most 1/4, it's less likely for the same wrong result to dominate the outcomes across multiple runs, unless it's actually the correct result or there's an unusually high chance of that specific incorrect result compared to others.

Analysis: The effectiveness of this approach depends on the number of executions n and the distribution of correct vs. incorrect results. The law of large numbers suggests that as n increases, the observed frequencies of

outcomes should converge to their true probabilities. Therefore, by choosing n to be sufficiently large, we can increase the likelihood that the correct result will be observed more frequently than any incorrect ones, given the assumptions about the probabilities.

However, it's important to note that while this strategy can significantly improve the probability of arriving at the correct result, it cannot guarantee correctness. The algorithm's effectiveness is also contingent on the assumption that incorrect results are distributed in such a way that no single incorrect result is disproportionately likely compared to others (beyond the 1/4 limit).

In summary, leveraging the probabilistic guarantees about the original algorithm A, we can indeed design a more useful randomized algorithm for F by running A multiple times and using a majority or frequency-based decision rule to determine the most likely correct result. This method improves the overall reliability of the computation at the cost of additional executions of A.

Question 2.4.58 [DARA_Hor]

One observes that the protocol UMC is based on a nondeterministic protocol that simply guesses the position j where x and y differ, and then verifies if its guess was correct (i.e., if really xj != yj). We have converted this nondeterministic protocol into an MC protocol by assigning some probabilities to nondeterministic decisions about acceptance and rejection in situations in which the protocol does not know the right answer. The idea is to accept with probability less than, but close to, 1/2, and to reject with the complementary probability. The probability to accept in this uncertain situation must be so close, from below, to 1/2 that one computation accepting with certainty brings the overall probability of acceptance above 1/2. Can one apply this idea for converting any nondeterministic algorithm to an equivalent MC algorithm of the same efficiency?

Answer 2.4.58

The approach described for the UMC protocol involves transforming a nondeterministic algorithm into a Monte Carlo (MC) algorithm by assigning probabilities to the nondeterministic choices. Specifically, in cases where the algorithm doesn't know the correct answer, it accepts with a probability just under 1/2 and rejects with the complementary probability. This strategy ensures that even one computation accepting with certainty can increase the overall probability of acceptance above 1/2. The question is whether this idea can be applied to convert any nondeterministic algorithm into an equivalent MC algorithm with the same efficiency.

In theory, the idea of converting nondeterministic algorithms to MC algorithms by assigning probabilities to nondeterministic decisions is a powerful concept. However, applying this strategy effectively across all nondeterministic algorithms seems hard and involves several challenges and limitations:

1. **Correctness Probability:** The essence of MC algorithms is that they must provide a correct answer with a probability greater than 1/2. This is relatively straightforward in scenarios with a binary decision (e.g., equality testing where xj != yj), but may be more complex in algorithms with multiple correct answers or paths. Ensuring that the overall probability of correctness exceeds 1/2, considering all possible outcomes and paths, can become challenging for more complex nondeterministic algorithms.
2. **Efficiency and Probability Assignment:** The efficiency of the converted MC algorithm heavily relies on how probabilities are assigned to the nondeterministic choices. For the conversion to maintain the same efficiency as the original nondeterministic algorithm, these probabilities must be assigned in a way that does not significantly increase the expected number of computations or the complexity of decision-

making. This can be difficult to achieve, especially for algorithms where the number of nondeterministic paths is large or where the structure of the problem does not lend itself to simple probabilistic decisions.

3. **Dependence on Problem Structure:** The effectiveness of converting a nondeterministic algorithm to an MC algorithm using this method depends on the specific structure of the problem the algorithm is solving. Some problems might naturally fit into this framework, especially those with clear binary decisions or where incorrect decisions lead to quick rejections. However, for algorithms solving more complex problems with multiple layers of decisions or where incorrect choices do not immediately lead to failure, designing an equivalent MC algorithm that retains the efficiency of the nondeterministic version can be more complicated.
4. **Impact on Overall Probability of Acceptance:** The strategy of accepting with a probability just under 1/2 and rejecting with the complementary probability works well when the outcome of the algorithm hinges on a single nondeterministic guess, as in the UMC protocol example. However, for algorithms where multiple correct guesses or decisions are required to reach a correct conclusion, ensuring that the overall probability of acceptance remains above 1/2 after multiple probabilistic decisions might require a more nuanced approach to probability assignment.

In conclusion, while the idea of converting nondeterministic algorithms into MC algorithms by probabilistically determining nondeterministic decisions is intriguing and can be applied to certain types of algorithms, its general applicability to all nondeterministic algorithms with the same efficiency is not guaranteed. The success of such a conversion depends on the nature of the algorithm, the problem it solves, and the ability to design a probabilistic decision-making process that enhances the likelihood of correctness without compromising efficiency. This approach requires careful consideration of the algorithm's structure and the probabilities involved to ensure that the converted MC algorithm is both correct and efficient.

Question Exercise 1.4 [RA_Mot]

Let 0 < £2 < £1 < 1. Consider a Monte Carlo algorithm that gives the correct solution to a problem with probability at least 1- £1, regardless of the input. How many independent executions of this algorithm suffice to raise the probability of obtaining a correct solution to at least 1- £2, regardless of the input?

Answer 1.4

To address this question, we need to calculate how many independent executions of the Monte Carlo algorithm are required to increase the probability of obtaining a correct solution to at least 1–£2, given that a single execution has a probability of at least 1-£1 of being correct.

Let's denote:

- Ps as the success probability in one run, which is at least 1 - ε1.
- n as the number of independent runs of the algorithm.
- Pf as the failure probability in one run, which is at most ε1.

To achieve an overall success probability of at least 1 - ε2, we consider the opposite scenario, which is simpler to calculate. The chance of all runs failing (not getting the correct solution in any of the runs) should be at most ε2.

The failure probability in each run is independent and at most ε1. So, the probability of all n runs failing is at most ε1 raised to the power of n.

We set this to be less than or equal to ε2 to solve for n:

ε1^n <= ε2

Taking the logarithm of both sides gives us:

n * log(ε1) <= log(ε2)

Since log(ε1) is negative (because 0 < ε1 < 1), when we divide by log(ε1) to solve for n, the inequality direction changes:

n >= log(ε2) / log(ε1)

This formula tells us the minimum number of independent runs needed to ensure the success probability is at least 1 - ε2. Since n must be a whole number, if the result is not an integer, we round up to the nearest whole number.

Let's calculate n for specific values of ε1 and ε2 to demonstrate this.

Given the values of ε1 = 0.2 and ε2 = 0.01, the minimum number of independent runs needed is 3. This means running the Monte Carlo algorithm independently 3 times is sufficient to increase the chance of getting the correct solution to at least 1 - ε2, regardless of the input.

Python code snippet for the calculation of n, regarding ε1 and ε2:

```
import math

# Example values for epsilon_1 and epsilon_2
epsilon_1 = 0.2  # Example value for ε1
epsilon_2 = 0.01  # Example value for ε2

# Calculate the minimum number of executions n
n = math.log(epsilon_2) / math.log(epsilon_1)

# Since n must be an integer, we round up to the nearest whole number
n = math.ceil(n)

print(n)
```

Question Exercise 1.6 [RA_Mot]

If X were to range over all integers having value at most m-1 (possibly including negative integers), how would the statement and proof of Theorem 1.3 change?

Answer 1.6

The theorem and its proof are predicated on the assumption that X is chosen from a distribution where E[X] is greater than or equal to g(n), and g(y) is non-decreasing. The theorem provides an upper bound on the expected number of steps E[T] that it takes a particle to reach a certain position.

If X can now take on negative values (which would correspond to the particle moving backwards), the expectation calculation would have to consider these possibilities. Specifically, it could take longer for the

particle to reach the position 1 since there's now a chance that the particle moves away from the target, not just towards it.

The original proof relies on the fact that after the first step, the particle is always closer to the position 1. If X can be negative, this is no longer guaranteed. The new proof would need to account for the potential for the particle to move in either direction.

The adjustment to the proof would require considering the additional expected distance the particle might have to travel due to possibly moving away from the position 1. Essentially, the expected number of steps E[T] would have to include the possibility of steps taken in the wrong direction.

To accommodate negative values of X, the proof would thus become more complex, as it would also have to account for the probabilities of moving backwards and the additional expected steps this would entail. The provided upper bound function f(m) might have to be modified to account for this, as the existing form assumes that each step will always reduce the remaining distance to the target by a non-negative amount. The new function f'(m) for the modified scenario would likely be a function that grows more slowly as m increases compared to the original f(m), since it would have to factor in the possibility of steps that increase the distance to the target.

Therefore, the statement of Theorem 1.3 would have to be changed to reflect that the particle may move away from the target, and the upper bound f(m) may have to be adjusted accordingly. The proof would need to incorporate the additional complexity of potentially moving backwards, which would alter the calculations for the expected number of steps E[T]. This would require integrating over the probabilities of moving backwards as well as forwards.

Given the particle can move backward or stay in place (in the case where X=0), it’s not guaranteed to make progress towards the target in each step. Thus, the expectation of the number of steps to reach the target would generally increase, and the upper bound provided by f(m) would have to be recalculated to incorporate these new scenarios.

The new proof would have to include terms accounting for the steps backward and could potentially introduce a series of conditions or cases in the induction step, where one would have to consider the expected number of steps from the new position n−X where X could now be negative, zero, or positive.

In sum, the new function f'(m) that would replace f(m) in the statement of Theorem 1.3 would reflect a more complicated path to the target, with the probability of backward steps decreasing the efficiency of progress towards the target. This could involve a combination of convolutions of the distributions of forward and backward steps to find the new expectation.

The inequality would no longer be as straightforward as E[T]≤f(n) but would likely have to be replaced with a more complex inequality that captures the additional variance introduced by the possibility of backward movement. The final form of this inequality would depend on the specific distributions of step sizes both forward and backward.

If X can range over all integers with values at most m−1, including negative integers, the theorem would need to be redefined to reflect the fact that the particle can move backwards, which was not considered in the original theorem. Here's how the theorem might be restated to reflect the change:

Modified Theorem 1.3: Let T be the random variable denoting the number of steps in which the particle reaches the position 1. If X can take on negative values, representing steps backwards, then E[T] may no longer be simply bounded by $\int_1^m f(x)\,dx/g(x)$ as in the original theorem. Instead, E[T] will be a more complex function that accounts for the possibility of moving both towards and away from the target position.

The modified theorem would need to include terms or conditions that reflect the bi-directional movement. The expectation E[T] would likely involve an integral or sum that accounts for the full range of X, considering the backward steps as well. The modified theorem might look something like this:

Modified Theorem Statement: Let T be the random variable denoting the number of steps in which the particle reaches the position 1. Then E[T] is bounded above by a function f'(m) that incorporates the probability of backward and forward steps. The function f'(m) would be determined based on the distribution of X and the probability of moving in either direction from any given position.

The exact form of f'(m) would depend on the specific distributions of step sizes. If the distribution is symmetric around zero or has a mean of zero, f'(m) could potentially resemble the original f(m), but scaled by a factor that reflects the inefficiency introduced by the possibility of moving backwards. If the distribution is not symmetric or has a negative mean, f'(m) would likely be a more complex function that reflects the skewed likelihoods of forward versus backward movement.

The proof of the modified theorem would change in a way to involve a new induction step that carefully accounts for the two possibilities at each stage: moving closer to the target or moving further away. This modified proof would need to handle the added complexity of backtracking steps and their impact on the expected time to reach the target.

Question Exercise 1.8 [RA_Mot]

Show that if there is a polynomial reduction from L1 to L2, then L2 $\in$ P implies that L1 $\in$ P.

Answer 1.8

If we can reduce language L1 to language L2 in polynomial time, and if L2 is in P (the class of problems solvable in polynomial time), then L1 must also be in P.

The reasoning we can back this statement with:

1. **Polynomial-Time Reduction:** A polynomial reduction from L1 to L2 means that there exists a polynomial-time computable function f such that for any string x, x is in L1 if and only if f(x) is in L2. This function f transforms instances of L1 into instances of L2.
2. **L2 is in P:** Since L2 is in P, there exists a deterministic Turing machine M2 that decides L2 in polynomial time. This means for any input y, M2 will correctly determine whether y is in L2 or not within a time bound that is a polynomial function of the length of y.
3. **Deciding L1 Using L2:** To decide whether an input x is in L1, we can use the polynomial-time reduction function f to convert x into f(x), and then run the machine M2 on f(x) to decide whether f(x) is in L2.
4. **Time Complexity:** The reduction takes polynomial time, and running M2 also takes polynomial time. The composition of two polynomial-time operations is still polynomial time. Therefore, the overall process to decide L1 is also polynomial time.
5. **L1 is in P:** Since we can decide whether any string x is in L1 using a polynomial-time reduction followed by a polynomial-time decision procedure for L2, it follows that L1 is also in P.

In conclusion, if we can reduce L1 to L2 in polynomial time and L2 can be decided in polynomial time, then we can also decide L1 in polynomial time, which implies that L1 is in P.

As I started solving DARA_hor first and it had exercises, when I went for RA_Mot I started with exercises and solved until I found out that it does not have exercise 1.11, and I saw that in this one you meant probably Problems by saying تمرين so I go for the problems and see how many I can solve in the remaining time that I have!

Question Problem 1.11 [RA_Mot]

Verify the following inclusions:

$P \subseteq RP \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP$.

It is not known whether these inclusions are strict.

Answer 1.11

$P \subseteq RP$: P (Polynomial time) is the class of decision problems that can be solved by a deterministic Turing machine in polynomial time. RP (Randomized Polynomial time) includes all problems for which there exists a randomized algorithm that runs in polynomial time and satisfies the following: If the answer is "no," it always returns "no"; if the answer is "yes," then it returns "yes" with a probability of at least 1/2. Every problem that can be solved deterministically in polynomial time can also be solved by a randomized algorithm in polynomial time (by simply not using randomness), so P is a subset of RP.

$RP \subseteq NP$: NP (Nondeterministic Polynomial time) is the class of decision problems for which, if the answer is "yes," there exists a "certificate" (or witness) that can be verified in polynomial time by a deterministic Turing machine. Since an RP algorithm has a probability of at least 1/2 of providing the correct "yes" answer (and can be run multiple times to boost this probability arbitrarily close to 1), any problem in RP also has a certificate that can be verified in polynomial time (the correct output of the RP algorithm), and therefore RP is a subset of NP.

$NP \subseteq PSPACE$: PSPACE is the class of decision problems that can be solved by a deterministic Turing machine using a polynomial amount of space. Since a nondeterministic Turing machine can be simulated by a deterministic Turing machine using a polynomial amount of space (by tracking all possible nondeterministic choices at once), NP is a subset of PSPACE.

$PSPACE \subseteq EXP$: EXP (Exponential time) includes problems that are solvable by a deterministic Turing machine in exponential time. Since polynomial space can be used to simulate exponential time (but not necessarily vice versa), PSPACE is a subset of EXP.

$EXP \subseteq NEXP$: NEXP (Nondeterministic Exponential time) includes problems that are solvable by a nondeterministic Turing machine in exponential time. Since deterministic algorithms are a special case of nondeterministic algorithms (where the nondeterministic choices are fixed), every problem that can be solved in exponential time by a deterministic Turing machine can also be solved by a nondeterministic Turing machine in exponential time, placing EXP within NEXP.

Question Problem 1.12 [RA_Mot]

Verify the following inclusions:

RP⊆BPP⊆PP.

It is not known whether these inclusions are strict.

Answer 1.12

RP ⊆ BPP: RP stands for Randomized Polynomial time, where an algorithm returns the correct answer with a probability of more than 1/2 if the answer is "yes," and always returns "no" if the answer is indeed "no." BPP stands for Bounded-error Probabilistic Polynomial time, which is the class of decision problems that can be solved by a randomized algorithm with an error probability of less than 1/3 (or any constant less than 1/2) for both "yes" and "no" answers, within polynomial time. Clearly, any problem in RP is also in BPP because an RP algorithm is a special case of a BPP algorithm where the "no" answers have zero error. Thus, RP is a subset of BPP.

BPP ⊆ PP: PP stands for Probabilistic Polynomial time and represents the class of decision problems for which there is a probabilistic Turing machine that accepts with a probability greater than 1/2 if the answer is "yes," and accepts with a probability less than or equal to 1/2 if the answer is "no." The key difference between BPP and PP is that in BPP, the error probability has to be bounded away from 1/2 by some fixed constant for both "yes" and "no" cases, whereas in PP, the probability only needs to be more than 1/2, even if it is infinitesimally more. Since BPP algorithms have a more stringent requirement on their error probability, every BPP algorithm meets the criteria for PP. Therefore, BPP is a subset of PP.

Question 1.18 [PC_Mit]

We have a function F: {0,...,n-1} -> {0,...,m-1}. We know that, for $0 \le x,y \le n - 1$, $F((x + y) \bmod n) = (F(x) + F(y)) \bmod m$. The only way we have for evaluating F is to use a lookup table that stores the values of F. Unfortunately, an Evil Adversary has changed the value of 1/5 of the table entries when we were not looking.

Describe a simple randomized algorithm that, given an input z, outputs a value that equals F(z) with probability at least 1/2. Your algorithm should work for every value of z, regardless of what values the Adversary changed. Your algorithm should use as few lookups and as little computation as possible.

Suppose I allow you to repeat your initial algorithm three times. What should you do in this case, and what is the probability that your enhanced algorithm returns the correct answer?

Answer 1.18

let's first devise a simple randomized algorithm that can output the correct value of F(z) with at least a 1/2 probability.

Algorithm:

1. Randomly pick an integer y in the range [0, n-1].
2. Compute $x = (z - y) \bmod n$. This calculation is done to find an x such that $x + y \equiv z \pmod n$.

3. Look up the values of F(x) and F(y) in the lookup table.
4. Calculate the sum F(x) + F(y) mod m. According to the function's properties, this sum should equal F(z) if neither F(x) nor F(y) has been tampered with.

Since 1/5 of the table has been altered, the probability that a lookup yields an incorrect value is 1/5. Therefore, the probability that a lookup gives the correct value is 1 - 1/5 = 4/5. When we pick two values from the table independently, the probability that both are correct is (4/5) * (4/5) = 16/25, which is less than 1/2.

To increase the probability of getting the correct answer, we perform the algorithm three times independently and take a majority vote on the results.

Enhanced Algorithm:

1. Run the initial algorithm three times independently to get three values: A1, A2, and A3.
2. If at least two of the values match, output that value. If all three are different, output one of the values at random or indicate failure.

Probability Analysis:

Let's calculate the probability that we get the correct value after three independent executions. The probability of getting the correct value in a single execution is at least 16/25.

1. The probability that all three executions return the correct result is (16/25)^3.
2. The probability that exactly two executions return the correct result is 3 * (16/25)^2 * (9/25) because there are three ways this can happen (correct, correct, incorrect or correct, incorrect, correct or incorrect, correct, correct).
3. We do not consider the cases where fewer than two executions are correct since we would not choose the correct value in the majority.

Now, let's add the probabilities from points 1 and 2 to get the total probability that the majority vote gives us the correct result.

For one trial, the probability of getting the correct answer is 4/5 for one lookup. For two lookups, since they are independent, we would square that probability (as mentioned above) to get the combined probability. However, this turned out to be incorrect as it's less than 1/2, which doesn't satisfy our requirement.

If we calculate the probability, we have:

1. Probability that both lookups are correct: $(4/5)*(4/5)=16/25$.
2. Probability that exactly one lookup is correct (either the first or the second): $2*(4/5)*(1/5)=8/25$.
3. Probability that both are wrong: $(1/5)*(1/5)=1/25$.

For the algorithm running three times, we want the probability that at least two of the three are correct. So let's calculate it:

- All three are correct:(16/25)^3.
- Exactly two are correct: $3*(16/25)^2*(9/25)$, since there are three different ways we could have two correct and one incorrect result.

Adding these two probabilities, we have the probability that our enhanced algorithm (majority vote after three trials) returns the correct answer.

Question 1.26 [PC_Mit]

Tic-tac-toe always ends up in a tie if players play optimally. Instead, we may consider random variations of tic-tac-toe.

(a) First variation: Each of the nine squares is labeled either X or O according to an independent and uniform coin flip. If only one of the players has one (or more) winning tic-tac-toe combinations, that player wins. Otherwise, the game is a tie. Determine the probability that X wins. (You may want to use a computer program to help run through the configurations.)

(b) Second variation: X and O take turns, with the X player going first. On the X player's turn, an X is placed on a square chosen independently and uniformly at random from the squares that are still vacant; O plays similarly. The first player to have a winning tic-tac-toe combination wins the game, and a tie occurs if neither player achieves a winning combination. Find the probability that each player wins. (Again, you may want to write a program to help you.)

Answer 1.26

Part (a)

To solve the first variation of tic-tac-toe, we can use the following approach:

- Each of the nine squares can be X or O, so there are 2^9 possible board configurations since each square has 2 options.
- We need to determine how many of these configurations are winning configurations for X. Since the coin flip is independent and uniform, each configuration has an equal chance of occurring.
- For X to win, there must be at least one line (row, column, or diagonal) with all X's and no line with all O's.
- There are 8 winning lines for X and 8 for O. A winning board for X excludes any board that has a winning line for O.

However, this becomes a bit complex because we need to avoid counting configurations that result in wins for both players, which isn't possible in a single game of tic-tac-toe. Thus, configurations where both players have a winning line should be excluded from the count.

Given the symmetry of the problem, the probability that X wins should be the same as the probability that O wins. We can calculate this probability by computing all possible winning configurations for X, making sure not to double-count configurations that would be winning for both players, and dividing by the total number of configurations.

To calculate this we should write a computer program that generates all possible board configurations, checks for winners, and counts the valid winning configurations for X:

```python
from itertools import product

# Generate all possible configurations of the board
# Each square can be either 'X' or 'O'
all_possible_boards = list(product('XO', repeat=9))

# Define winning combinations for tic-tac-toe
winning_combinations = [
    (0, 1, 2), (3, 4, 5), (6, 7, 8),  # rows
    (0, 3, 6), (1, 4, 7), (2, 5, 8),  # columns
    (0, 4, 8), (2, 4, 6)              # diagonals
]

# Function to check if a player ('X' or 'O') has won in the given board configuration
def has_won(board, player):
    return any(all(board[pos] == player for pos in combo) for combo in winning_combinations)

# Count the number of winning boards for 'X' and 'O'
x_wins = 0
o_wins = 0

# Iterate through all possible board configurations
for board in all_possible_boards:
    # Check for 'X' and 'O' wins
    x_win = has_won(board, 'X')
    o_win = has_won(board, 'O')

    # If 'X' wins and 'O' doesn't, increment the 'X' win count
    if x_win and not o_win:
        x_wins += 1
    # If 'O' wins and 'X' doesn't, increment the 'O' win count (not needed for this exercise
    elif o_win and not x_win:
        o_wins += 1

# The total number of configurations
total_configurations = len(all_possible_boards)

# The probability that X wins
probability_x_wins = x_wins / total_configurations

print("Probability that x wins: ", probability_x_wins)

```

```
Probability that x wins:  0.38671875
```

Part (b)

For the second variation, we approach the problem by simulation because the number of possible game outcomes becomes quite large due to the various paths the game can take after each move.

1. Simulate a large number of games where X and O are placed randomly in turns.
2. For each game, check if there's a winner or if the game ends in a tie.
3. Record the number of wins for X and O.
4. The probability of X or O winning is the number of wins for that player divided by the total number of games simulated.

We would expect X to have a slight advantage since X goes first.

To implement these simulations, we would write a program to generate random games and tally the results. This would give us an empirical probability for X and O winning in each of the variations:

```
import random

def check_winner(board):
    # Winning combinations
    wins = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8), (0, 4, 8), (2,
    for a, b, c in wins:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return board[a]
    return None

def simulate_game():
    # Initialize a blank tic-tac-toe board
    board = [' '] * 9
    # X goes first
    turn = 'X'

    while ' ' in board:
        # Generate a list of valid moves
        valid_moves = [i for i in range(9) if board[i] == ' ']
        # If no valid moves, it's a tie
        if not valid_moves:
            break
        # Randomly select a move for the current player
        move = random.choice(valid_moves)
        board[move] = turn
        # Check for a winner
        winner = check_winner(board)
        if winner:
            return winner
        # Switch turns
        turn = 'O' if turn == 'X' else 'X'
    return 'tie'

# Run the simulation for a number of games and count the results
def run_simulation(num_games):
    results = {'X': 0, 'O': 0, 'tie': 0}

    for _ in range(num_games):
        winner = simulate_game()
        results[winner] += 1

    return results

# Let's simulate 100,000 games
num_games = 100000
simulation_results = run_simulation(num_games)

# Calculate the probabilities for each outcome
probabilities = {k: v / num_games for k, v in simulation_results.items()}
print("The probability that each player wins: ", probabilities)

```

```
The probability that each player wins:  {'X': 0.58323, 'O': 0.29036, 'tie': 0.12641}
```

Question 2.22 [PC_Mit]

Let a1, a2, ..., an be a list of n distinct numbers. We say that ai and aj are inverted if i < j but ai > aj. The Bubblesort sorting algorithm swaps pairwise adjacent inverted numbers in the list until there are no more inversions, so the list is in sorted order. Suppose that the input to Bubblesort is a random permutation, equally likely to be any of the n! permutations of n distinct numbers. Determine the expected number of inversions that need to be corrected by Bubblesort.

Answer 2.22

To determine the expected number of inversions that need to be corrected by Bubblesort for a list of n distinct numbers, we can think about the problem as follows:

An inversion is a pair (i, j) such that i is less than j and ai is greater than aj. When the list is a random permutation of the numbers 1 to n, each pair of positions (i, j) with i less than j is equally likely to be an inversion or not. Since there are no duplicate numbers, each pair is independent of the others.

There are n(n-1)/2 such pairs in total, which is the number of ways to choose 2 items from n, without considering the order. This is because we have n-1 choices for aj when i is 1, n-2 choices when i is 2, and so on, down to 1 choice when i is n-1.

The probability that any given pair is an inversion in a random permutation is 1/2, because for any two different numbers ai and aj, there are two possible orders, and one of them is an inversion.

Therefore, the expected number of inversions in a random permutation of n numbers is half the total number of pairs:

E(number of inversions) = n(n-1)/2 * 1/2 = n(n-1)/4

So, the expected number of inversions for Bubblesort to sort a random permutation of n distinct numbers is n(n-1)/4.

Question 2.23 [PC_Mit]

Linear insertion sort can sort an array of numbers in place. The first and second numbers are compared; if they are out of order, they are swapped so that they are in sorted order. The third number is then placed in the appropriate place in the sorted order. It is first compared with the second; if it is not in the proper order, it is swapped and compared with the first. Iteratively, the kth number is handled by swapping it downward until the first k numbers are in sorted order. Determine the expected number of swaps that need to be made with a linear insertion sort when the input is a random permutation of n distinct numbers.

Answer 2.23

To calculate the expected number of swaps needed for insertion sort, we need to consider each element's position in the list, starting from the second element. For an element in position k, the expected number of swaps is the expected number of elements greater than it among the first k-1 elements. In a random permutation, we expect about half of these k-1 elements to be greater than our target element.

The expected number of swaps for each position k from 2 to n can be calculated as (k-1)/2. To find the total expected number of swaps for the entire list, we sum up the expected number of swaps for each element:

Expected Swaps = sum of (k-1)/2 for all k from 2 to n

This sum gives us the formula:

Expected Swaps = 1/2 + 2/2 + 3/2 + ... + (n-1)/2

This simplifies to:

Expected Swaps = 1/2 * sum of k from 1 to n-1

The sum of the first n-1 integers is (n-1)n/2, so the expected number of swaps is:

Expected Swaps = 1/2 * (n-1)n/2 = (n-1)n/4

This is the formula we can use to calculate the expected number of swaps for any n. If we input this formula into a calculator, it will give us the expected number of swaps for insertion sort on a list of n distinct numbers in a random permutation.

For example to calculate the expected number of swaps needed for insertion sort on list of 10 values we have:

```
# Calculate the expected number of swaps needed for insertion sort
def expected_swaps(n):
    # The first element doesn't need to be swapped, so we start from the second element (k=2)
    total_swaps = sum((k - 1) / 2 for k in range(2, n + 1))
    return total_swaps


n = 10
print("The expected number of swaps needed: ", expected_swaps(n))

```

```
The expected number of swaps needed:  22.5
```