CS 124 Homework 6: Spring 2024

Collaborators: Rowan Wang, Sophie Zhu, Collin Fan, Lavik Jain, Kathryn Harper, Davin Jeong, Charlie Chen, Annabel Ma

No. of late days used on previous psets: 6
No. of late days used after including this pset: 5

Homework is due Wednesday Apr 10 at 11:59pm ET. Please remember to select pages when you submit on Gradescope. Each problem with incorrectly selected pages will lose 5 points.

Try to make your answers as clear and concise as possible; style may count in your grades. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan your papers to turn them in.

Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential-time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or no credit. Again, try to make your answers clear and concise.

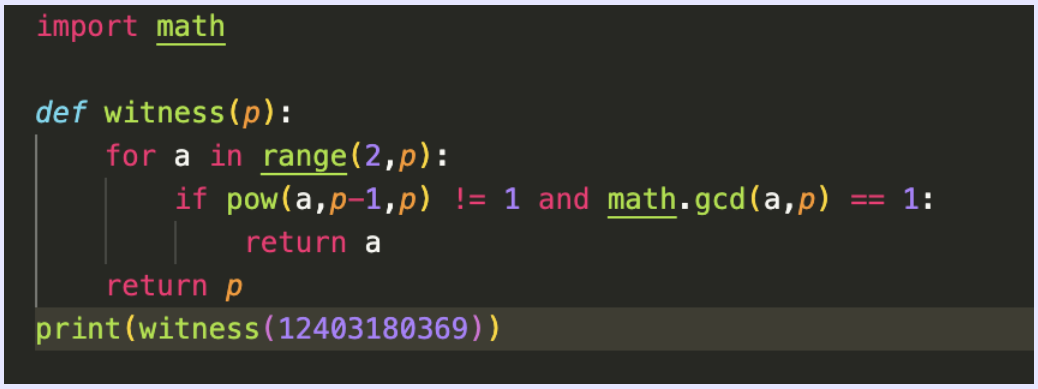
Problems

1. (a) **(10 points)** Prove that 12403180369 is composite by finding a witness in the form of a positive integer $a < 12403180369$ such that $a^{12403180369-1} \neq 1 \pmod{12403180369}$. Give any information necessary to show that your witness in fact witnesses. (Note: do not use a factor as a witness! Sure, 12403180369 is small enough that you can exhaustively find a factor, but this question is about your knowledge of the primality-testing algorithm.)

Proof. According to Fermat's little theorem, if p is a prime number and a is not a multiple of p, then $a^{p-1} = 1 \mod p$. Thus, to prove that 12403180369 is not a prime (composite), we will find a positive integer a which is not a multiple of 12403180369 such that $a^{12403180369-1} \neq 1 \pmod{12403180369}$.

To do this, we will implement a program in python. This algorithm will check if

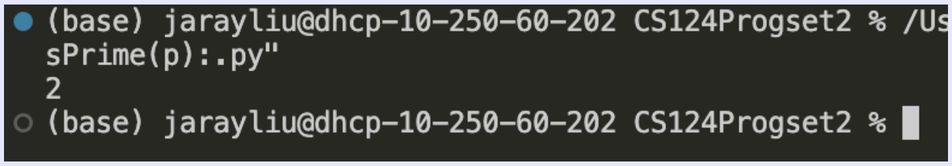
$a^{p-1} \not\equiv 1 \mod p$ and $\gcd(a,p) = 1$ (which implies that a is not a multiple of p) for all positive integers $1 < a < p$, returning the valid witness a if any a satisfies the above conditions.



```python
import math

def witness(p):
    for a in range(2,p):
        if pow(a,p-1,p) != 1 and math.gcd(a,p) == 1:
            return a
    return p
print(witness(12403180369))
```

When this code is executed, the output is below:



```
● (base) jarayliu@dhcp-10-250-60-202 CS124Progset2 % /Us
  sPrime(p):.py"
  2
○ (base) jarayliu@dhcp-10-250-60-202 CS124Progset2 %
```

Thus, p is composite and a valid witness is $a = 2$ since $\gcd(2, 12403180369) = 1$ and $2^{12403180369-1} = 1464795051 \mod 12403180369$, meaning that $2^{12403180369-1} \neq 1 \mod 12403180369$. □

(b) **(10 points)** The number 63973 is a Carmichael number. Prove that it is composite by finding a witness in the form of a nontrivial square root of 1.

Proof. According to theorem 2 of lecture notes 17, if n is prime, the only solutions to $a^2 \equiv 1 \mod n$ are $a = 1, a = -1$. Therefore, if there exists an a such that $a^2 \equiv 1 \mod n$ and $a \neq 1, a \neq -1$, then n is not prime.

It follows that to prove that 63973 is composite, we must find a witness a such that $a^2 \equiv 1 \mod 63973$ and $a \neq 1, a \neq -1$ (we find a witness a in the form of a nontrivial square root of 1).

Consider witness $a = 6252$. We have $6252^2 = 39087504 \equiv 1 \mod 63973$ and $6252 \neq 1$ and $6252 \neq -1$. Thus we have found a nontrivial square root of 1 which implies that 63973 is composite. □

(You can find the answers by whatever method you like, but we recommend writing code. You don't need to submit your code.)

2. Consider the task of generating a uniformly random number from 1 to N together with its prime

factorization. One naive approach would be to simply generate a number from 1 to N and then factor it, but this would not be computationally efficient. In this problem we will explore an alternative approach. The algorithm is as follows:

Algorithm 1

```
1: Input: $N$
2: $m \leftarrow N$
3: $i \leftarrow 1$
4: while $m > 1$ do
5:     Let $s_i$ be a uniformly random number from 1 to $m$
6:     $m \leftarrow s_i$
7:     $i \leftarrow i + 1$
8: end while
9: Remove all the non-prime $s_i$'s and let $r$ be the product of all prime $s_i$'s generated above.
10: if $r \leq N$ then
11:     With probability $r/N$, return $r$ together with the list of all prime $s_i$'s generated above. Otherwise, restart the algorithm (i.e. go back to line 2).
12: else
13:     go back to line 2
14: end if
```

Algorithm 1 runs in expected polynomial time in the number of bits needed to describe N (you need not prove this, but in particular line 9 can be implemented in polynomial time, and optional part (e) below shows why the number of iterations of the whole algorithm till it outputs something is also polynomial). Your goal is to prove that the algorithm outputs a random number in $\{1, \dots, N\}$.

(a) (**5 points**) Prove that the probability that exactly t of the s_i's are equal to N is given by $\frac{1}{N^t}(1 - 1/N)$.

Proof. By the construction of the algorithm, m is set to s_i in every iteration of the while loop. Thus, to even have the possibility that $s_i = N$, we must have $s_{i-1} = N$. Otherwise, if $s_{i-1} \neq N$, then intuitively $s_{i-1} < N$ and the interval we select s_i from would shrink to not include N. This then implies that the moment an s_i is assigned a number less than N, the subsequent $s_{i+1}, s_{i+2}, \ldots$ can never be assigned N. Therefore, in order for t of the s_i's to be equal to N, then the first t of the s_i's must be equal to N.

Then, in order for *exactly* t of the s_i's to be equal to N, after $s_t = N$ is picked, the algorithm must choose $s_{t+1} < N$ so that the algorithm will never again randomly choose an $s_i = N$. Since the first t of the s_i's are uniformly randomly selected from the interval $\{1, \ldots, N\}$, the probability that the first t of the s_i's are all equal to N is $(\frac{1}{N})^t$. In addition, the probability that $s_{t+1} < N$ is the probability of uniformly randomly choosing a number from $1 \ldots N$ which is not equal to N. Note that this probability is $1 - \frac{1}{N}$, by the complement property. Therefore, the probability that *exactly* t of the s_i's is equal to N is the product of these probabilities, which is:

$$(\frac{1}{N^t})(1-\frac{1}{N})$$

□

(b) (**10 points**) Let $t_2,\dots,t_N$ be any nonnegative integers. Generalize your proof in the previous part to derive an expression for the probability that exactly t_a of the s_i's are equal to a for all $a \in \{2,\dots,N\}$. (In particular you should get a very simple expression for the probability that there is at least one i such that $s_i = a$.)

Proof. We will generalize our proof in part (a) to derive an expression for the probability that exactly t_a of the s_i's are equal to a for all $a \in \{2,\dots,N\}$. Crucially, if our algorithm ever chooses an $s_i = a$, then we apply the same logic as part (a) to find the probability that exactly t_a of the s_i's are equal to a. This uses the fact that immediately after we hit the first $s_i = a$, we must hit a exactly $t_a - 1$ more times in a row (since once $s_i < a$, subsequent $s_{i+1} \dots \neq a$). This probability is $\frac{1}{a^{t_a-1}}$. Then, after this sequence of t_a hits, we must have select an $s_i \neq a$ so that we never again can get an $s_i = a$, ensuring we have exactly t_a of the s_i's equal to a. This probability is $(1-\frac{1}{a})$. Thus,

$$\Pr[\text{exactly } t_a \text{ of the } s_i\text{'s equal to } a] = \Pr[\text{at least one } s_i = a] * \frac{1}{a^{t_a-1}} * (1-\frac{1}{a})$$

Therefore, to derive the desired expression, we must find Pr[at least one $s_i = a$] for all $a \in \{2,\dots,N\}$. We will use the hints TF Eric provided on Ed. First, note that since $a \in \{2,...,N\}$ and the s_i's must go down from N to 1, then there must exist some s_i which "crossed" over a. Thus, let $P(x)$ be the probability of hitting a $s_i = a$ sometime later given the condition that we just picked $s_i = x$. Naturally, $P(a) = 1$, since we have just picked an $s_i = a$. In addition, $P(i < a) = 0$ since once an s_i has crossed a, we can never hit a again. Thus, we can derive an expression for $P(x)$ for any $x > a$:

$$P(x) = \frac{1}{x}P(x) + \frac{1}{x}P(x-1) + \dots + \frac{1}{x}P(a+1) + \frac{1}{x}P(a) + \frac{1}{x}P(i<a) + \dots + \frac{1}{x}P(1)$$

$$P(x) = \frac{1}{x}P(x) + \frac{1}{x}P(x-1) + \dots + \frac{1}{x}P(a+1) + \frac{1}{x} + 0$$

We hypothesize that $P(x) = \frac{1}{a}$ for any $x > a$. We will prove this with strong induction. Our base case is when $x = a+1$. Using our formula, we see:

$$P(a+1) = \frac{1}{a+1}P(a+1) + \frac{1}{a+1}$$

$$P(a+1) = \frac{1}{a}$$

Thus our base case is proved correct. In our inductive step, we assume that for any $x = a+1, a+2, ..., a+b$, $P(x) = \frac{1}{a}$. We will show that this implies $P(a+b+1) = \frac{1}{a}$.

$$P(a+b+1) = \frac{1}{a+b+1}P(a+b+1) + \frac{1}{a+b+1}P(a+b) + \dots + \frac{1}{a+b+1}P(a+1) + \frac{1}{a+b+1}$$

$$(a+b)P(a+b+1) = P(a+b) + ... + P(a+1) + 1$$

$$(a+b)P(a+b+1) = 1 + \sum_{i=a+1}^{a+b} P(i)$$

By our inductive hypothesis,

$$(a+b)P(a+b+1) = 1 + \sum_{i=a+1}^{a+b} \frac{1}{a}$$

$$(a+b)P(a+b+1) = 1 + \frac{b}{a}$$

$$P(a+b+1) = \frac{1}{a+b} * \frac{a+b}{a}$$

$$P(a+b+1) = \frac{1}{a}$$

Therefore, we have proven inductively that the probability that we have at least one $s_i = a$ for any value of $m > a$ is $\frac{1}{a}$. Thus, the probability that there exists at least one i such that $s_i = a$ is simply $\frac{1}{a}$. Combining this with our earlier expression:

$$\Pr[\text{exactly } t_a \text{ of the } s_i\text{'s equal to } a] = \Pr[\text{at least one } s_i = a] * \frac{1}{a^{t_a - 1}} * (1 - \frac{1}{a})$$

We now have:

$$\Pr[\text{exactly } t_a \text{ of the } s_i\text{'s equal to } a] = \frac{1}{a} * \frac{1}{a^{t_a - 1}} * (1 - \frac{1}{a})$$

$$\Pr[\text{exactly } t_a \text{ of the } s_i\text{'s equal to } a] = \frac{1}{a^{t_a}} * (1 - \frac{1}{a})$$

□

(c) (**10 points**) Let r be a particular number with prime factorization $r = 2^{t_2} \cdot 3^{t_3} \cdots p^{t_p}$. Prove that the probability that the r computed in line 9 of the algorithm equals this particular choice of r is given by

$$\frac{1}{r} \cdot \prod_{2 \le p \le N \text{ prime}} \left(1 - \frac{1}{p}\right) \tag{1}$$

Proof. In part (b), we proved an expression for the probability that exactly t_a of the s_i's are equal to a for all $a \in \{2, ..., N\}$. Intuitively, we can express the probability that our particular choice of r is computed in line 9 of the algorithm as the product of the probabilities that exactly t_2 of the s_i's are 2, exactly t_3 of the s_i's are 3, and so on, based on the prime factorization of r. Note that for the s_i's that are prime numbers from 2...N which are not part of the prime factorization of r, we specify $t_i = 0$. In addition, note that for all primes $2, 3, ..., p$ are in the interval $\{2, ..., N\}$ since the algorithm can never chooses s_i greater than N. Thus, our probability expression from (b) holds for all primes $2 \le 2, 3, ..., p \le N$.

Thus, since any number is uniquely determined by its prime factorization, to have the number computed in line 9 to be our particular choice of r, we want exactly t_i occurrences of prime number $p_i \in \{2,3,...,p\}$. Recall that the probability that exactly t_a of the s_i's are equal to a for all $a \in \{2,...,N\}$ is:

$$\Pr[\text{exactly } t_a \text{ of the } s_i\text{'s are equal to } a] = (\frac{1}{a^t})(1-\frac{1}{a})$$

Therefore, the probability that number computed in line 9 is our particular choice of r is:

$$(\frac{1}{2^{t_2}})(1-\frac{1}{2}) * (\frac{1}{3^{t_3}})(1-\frac{1}{3}) * ... * (\frac{1}{p^{t_p}})(1-\frac{1}{p})$$

$$(\frac{1}{2^{t_2} * 3^{t_3} * ... * p^{t_p}}) * (1-\frac{1}{2}) * (1-\frac{1}{3}) * ... * (1-\frac{1}{p})$$

Since $r = 2^{t_2} * 3^{t_3} * ... * p^{t_p}$ and $2,3,...,p$ are the primes between 2 and N inclusive, we can simplify to:

$$(\frac{1}{r}) * (1-\frac{1}{2}) * (1-\frac{1}{3}) * ... * (1-\frac{1}{p})$$

$$(\frac{1}{r}) * \prod_{2 \le p \le N \text{ prime}} \left(1-\frac{1}{p}\right)$$

Thus, we have proved that the probability that the r computed in line 9 of the algorithm equals this particular choice of r is

$$\frac{1}{r} \cdot \prod_{2 \le p \le N \text{ prime}} \left(1-\frac{1}{p}\right)$$

□

(d) (**5 points**) Conclude that if this algorithm returns something, it must be a uniformly random number from 1 to N together with the list of its prime factors.

Proof. By the construction of the algorithm, we know that if any output r will be such that $r \le N$. We also know that $r \ge 1$ since all s_i's are lower bounded by 1. Furthermore, we are given that we return a number r with probability $\frac{r}{N}$ in line 11. Thus, the probability that the algorithm outputs a given r is the product of the probability $\frac{r}{N}$ and the probability that we generate r *in the first place*, since these two probabilities are independent. Note that the latter probability we derived in part (c). Therefore, this is:

$$\Pr[\text{output is a given } r] = \frac{r}{N} * \frac{1}{r} \cdot \prod_{2 \le p \le N \text{ prime}} \left(1-\frac{1}{p}\right)$$

$$\Pr[\text{output is a given } r] = \frac{1}{N} \cdot \prod_{2 \le p \le N \text{ prime}} \left(1-\frac{1}{p}\right)$$

Since this probability is not dependent on r, this implies for any $1 \le r \le N$, the probability that r is outputted is:

$$\frac{1}{N} \cdot \prod_{2 \le p \le N \text{ prime}} \left(1 - \frac{1}{p}\right)$$

Therefore, since every r in the range 1 to N has the same probability of being outputted, then by definition our algorithm outputs a uniformly random number from 1 to N. Moreover, since the algorithm by definition outputs r along with the list of its prime factors, then we conclude that our algorithm outputs a uniformly random number from 1 to N together with the list of its prime factors. □

(e) (**0 points, optional**)[1] Prove that the expected number of times the algorithm restarts is $O(\log n)$. You may find Mertens' third theorem useful.

3. Given an undirected graph G, a *nice* 2*-coloring* is an assignment of colors from {red, blue} to each of the vertices such that for any triangle in the graph (i.e. three vertices which all have edges between each other), the vertices in the triangle are not all assigned the same color.

   Let G be a graph for which there exists a *proper* 3*-coloring*, that is, an assignment of colors from {red, blue, green} to each of the vertices such that no two vertices connected by an edge have the same color.

   (a) (**10 points**) Prove that G has a nice 2-coloring. (In fact G has many nice 2-colorings. You don't have to prove this, but understanding why this is so might help with the next part.)

Proof. Let us first consider the proper 3-coloring of G. By definition of proper 3-coloring, we know that no two vertices connected by an edge have the same color. This implies that for any triangle in the proper 3-coloring of G, all three vertices have different assigned colors. We will prove this by contradiction: assume for the sake of contradiction that there exists a triangle in the proper 3-coloring of G such that the vertices in the triangle do not all have different assigned colors. Then, since each vertex in the triangle is connected to the other 2 vertices by an edge (by definition of a triangle), then there must exist two vertices in the triangle connected by edge which have the same assigned color. However, this contradicts the property of proper 3-coloring. Thus, the statement is proved true.

Next, consider change all green vertices to blue vertices in the proper 3-coloring of G. Recall that we have proved that for any triangle in the proper 3-coloring of G, all three vertices are assigned different colors. This implies that every triangle in the proper 3-coloring of G has a red, a green, and a blue vertex. Then, when we change all green vertices to blue in the proper 3-coloring of G, it follows that every triangle in this new G must have a red, a blue, and another blue vertex. Therefore, for any triangle in this new graph G, the vertices in the triangle are not all assigned the same color. By definition, this means G has a nice 2-coloring.

[1] We won't grade this problem, but may use it for recommendations/TF hiring.

Thus, we have proved that if G has a proper 3-coloring, then G must have a nice 2-coloring. □

(b) (**15 points**) Give a randomized algorithm for finding this coloring that has expected runtime polynomial in the number of vertices of G. Determine the expected runtime of this algorithm (that is, analyze the expected amount of time that this algorithm takes before finding a nice 2-coloring.)

Proof. Consider an algorithm where every triangle in our graph is represented by a clause with three variables. Instead of true and false, each variable has three states representing the possible coloring's red, green, and blue. Thus, we initialize our algorithm by first finding all triangles in the graph through checking all $\binom{n}{3}$ groups of 3 vertices, creating clauses for every triangle we find. Then, to conclude the set up of our algorithm, we set all vertices in the entire graph (including the ones not in triangles) to random colors.

Now, let us define our algorithm below:

i. While some clause (triangle) is not nicely 2-colored (that is, if all vertices in the clause are monochromatic)

    A. Pick a random vertex in the clause and randomly flip its color to one of the other two colors.

ii. Return the current assignment of colors of vertices.

We note that returning the current assignment of colors of vertices also means we return the current assignment of colors of vertices not in triangles, in addition to the vertices in triangles. Also, note that since we proved there is a nice 2-coloring in part (a), we can use the while loop without risking infinite iterations of the loop.

To prove the correctness of our algorithm, we will prove that if and only if our algorithm returns an assignment of colors for the vertices in the graph, it is a nice 2-coloring. Note that from the construction of our algorithm, we immediately know that no triangle in the graph is monochromatic, since the while loop terminates and we return the current assignment of colors if and only if this condition is true. Also, note that the colors of the vertices not in any triangles does not affect proper 2-coloring, since the proper 2-coloring only cares about triangles are not monochromatic. Thus, when we return all the color assignments of all vertexes, our proper 2-coloring is not affected by the colors of vertices not in triangles (which we did not flip from the random color assignments). Therefore, our algorithm returns an assignment of colors for the vertices in the graph if and only if the assignment is a nice 2-coloring. In addition, our algorithm exhaustively produces a solution. This is proved immediately by the fact that our algorithm randomly flips vertices which lie in monochromatic triangles, implying that our algorithm is constantly inching towards a nice 2-coloring. In addition, there *must* exist a nice 2-coloring (which we proved in part (a)). Thus, since the while loop *will not terminate* until the nice 2-coloring property is satisfied and it is randomized, our algorithm could eventually go through every possible assortment of coloring's which we are guaranteed to have a proper 2-coloring in.

To prove the expected runtime of our algorithm, let A denote the current assignment of colors among vertices and let S denote the coloring assignments of all vertices in triangles in the proper 3-coloring of G. Furthermore, let T be the number of steps that our algorithm takes and let k be the number of vertices in A that match S. Note that every time we pick a random vertex in a monochromatic triangle and flip its color to one of the other colors, we have exactly 1 vertex already matching its color in S and 2 vertices not matching its color in S. This is because, as we previously discussed, in the proper 3 coloring S, every triangle contains one vertex of each color.

Thus, since we pick a vertex randomly in a monochromatic triangle, we flip a incorrectly colored vertex with $2/3$ chance and we flip a correctly colored vertex with $1/3$ chance. Obviously, if we flip a correctly colored vertex to any of the other colors, k decrements by 1 because we have just regressed. Now, if we flip an incorrectly colored vertex, we then have 2 choices of colors to flip to, one of which is the correct color. Intuitively, the other choice of color is also incorrect. Thus, at every color flip, k either decreases by 1 with probability $1/3$, remains the same (since we flip an incorrectly colored vertex to another incorrect color) with probability $2/3 * 1/2 = 1/3$, or increments by 1 with probability $2/3 * 1/2 = 1/3$. Therefore, let $T(i)$ denote the expected number of steps to walk from i to n, where i is the initial number of vertices in A which match S:

$$\begin{cases} T(n) = 0 \\ T(i) = \frac{1}{3}T(i-1) + \frac{1}{3}T(i) + \frac{1}{3}T(i+1) + 1 \\ T(0) = T(1) + 1 \end{cases}$$

Note that $T(n) = 0$ since if we start off with all n vertices in A matching S, we take no steps and return A. Also, $T(0) = T(1) + 1$ since if we start off with no matching vertices, then we take an arbitrary first step to get to $T(1)$. Rearranging $T(i) = \frac{1}{3}T(i-1) + \frac{1}{3}T(i) + \frac{1}{3}T(i+1) + 1$, we get:

$$\begin{cases} T(n) = 0 \\ T(i) = \frac{1}{2}T(i-1) + \frac{1}{2}T(i+1) + \frac{3}{2} \\ T(0) = T(1) + 1 \end{cases}$$

Plugging this into Wolfram Alpha, we get:

$$T(i) = \frac{1}{2}(-3i^2 + i + 3n^2 - n)$$

Therefore, the expected number of steps in the worst case when we start with no vertices in A matching S is $T(0) = \frac{1}{2}(3n^2 - n) = O(n^2)$.

Thus, our runtime is expected to be such: finding all triangles in the graph takes

$O(n^3)$ time to check all possible $\binom{n}{3}$ trios of vertices. Then, for each of the $O(n^2)$ steps, we must at worst check all $O(n^3)$ triangles in the graph to see if they are monochromatic, which takes $O(n^2 * n^3)$ time. Thus, the expected runtime of our algorithm in the number of vertices of G is $O(n^3 + n^2 * n^3) = O(n^5)$. □

(Hint 1: Consider an algorithm that picks an arbitrary starting coloring and while there exists a monochromatic triangle (i.e., a triangle where all vertices have the same color), picks a random vertex in the monochromatic triangle and flips its color. Now try an analysis similar to the random 2SAT algorithm.)

(Hint 2: You may find it helpful to partition the vertices of G into three parts, corresponding to the vertices that are assigned red, blue, and green respectively in the proper 3-coloring. Try keeping track of how far your 2-coloring is from this assignment over the course of your random walk.)

4. Suppose we have a random walk with boundaries 0 and n, starting at position i. As mentioned in class, this can model a gambling game, where we start with i dollars and quit when we lose it all or reach n dollars. Let W_t be our winnings after t games, where W_t is defined only until we hit a boundary (at which point we stop). Note that W_t is negative if we are at a position j with $j < i$; that is, we have lost money. If the probability of winning and the probability of losing 1 dollar each game are 1/2, then with probability 1/2, $W_{t+1} = W_t + 1$ and with probability 1/2, $W_{t+1} = W_t - 1$. Hence

$$E[W_{t+1}] = \frac{1}{2}E[W_t + 1] + \frac{1}{2}E[W_t - 1] = E[W_t],$$

where we have used the linearity of expectations at the last step. Therefore when the walk reaches a boundary, the expected winnings is $E[W_0] = 0$. We can use this to calculate the probability that we finish with 0 dollars. Let this probability be p_0. Then with probability p_0 we lose i dollars, and with probability $1 - p_0$ we gain $n - i$ dollars. Hence

$$p_0(-i) + (1 - p_0)(n - i) = 0,$$

from which we find $p_0 = (n - i)/n$.

Gossip Girl encounters this game in one of her classes at Constance Billard High School. Unfortunately, because it is high school, the game is not fair; instead, the probability of losing a dollar each game is 2/3, and the probability of winning a dollar each game is 1/3. Each student starts with i dollars on day 0 and plays a game on each day and gets to graduate from high school if they reach n dollars before going bankrupt. Bankrupt students don't get to play this game and stay in school for ever.

We wish to understand the probability that a student graduates (by extending the random walk analysis). In each of the following parts, including the optional one, you may assume the result from previous parts even if you did not prove it.

(a) **(5 points)** Show that $E[2^{W_{t+1}}] = E[2^{W_t}]$.

Since the game is now biased, the probability of winning and losing a dollar is not equal. Specifically, the probability that $W_{t+1} = W_t + 1$ is $\frac{1}{3}$ and the probability that $W_{t+1} = W_t - 1$ is $\frac{2}{3}$. Therefore, by the linearity of expectation, we express:

$$E[2^{W_{t+1}}] = \frac{1}{3}E[2^{W_t+1}] + \frac{2}{3}E[2^{W_t-1}]$$

Simplifying, we have:

$$E[2^{W_{t+1}}] = \frac{1}{3}E[2 * 2^{W_t}] + \frac{2}{3}E[\frac{1}{2}2^{W_t}]$$

Thus,

$$E[2^{W_{t+1}}] = E[2^{W_t}]$$

(b) **(5 points)** Use this to determine the probability of finishing with 0 dollars and the probability of finishing with n dollars when starting at position i.

Let $W_{t_{end}}$ be the winnings when the the walk reaches a boundary (ie. our winnings is n or 0). Since $E[2^{W_{t+1}}] = E[2^{W_t}]$ (from part (a)), this inductively implies that $E(2^{W_{t_{end}}}) = E(2^{W_0})$. Moreover, since at the first time step W_0, our winnings are $W_0 = 0$, then $E(2^{W_{t_{end}}}) = E(2^{W_0}) = 2^0 = 1$.

Let us denote p_0 as the probability that we lose i dollars when we hit a boundary and $1 - p_0$ as the probability that we gain $n - i$ dollars when we hit a boundary. Note that these probabilities are complements of each other since by definition of a boundary, we must have hit either position 0 or position n from a starting point i. It follows that the probability that $2^{W_{t_{end}}} = 2^{-i}$ is p_0 and the probability that $2^{W_{t_{end}}} = 2^{n-i}$ is $1 - p_0$. Thus,

$$E(2^{W_{t_{end}}}) = p_0(2^{-i}) + (1 - p_0)(2^{n-i})$$

$$1 = p_0(2^{-i}) + (1 - p_0)(2^{n-i})$$

$$1 = p_0 2^{-i} + 2^{n-i} - p_0 2^{n-i}$$

$$1 = p_0 2^{-i} + 2^n * 2^{-i} - p_0 2^n * 2^{-i}$$

$$2^i = p_0 + 2^n - p_0 2^n$$

$$2^i - 2^n = p_0(1 - 2^n)$$

$$p_0 = \frac{2^i - 2^n}{1 - 2^n}$$

Therefore, the probability that we end with 0 dollars (or equivalently, lose i dollars when we hit a boundary) is $p_0 = \frac{2^i - 2^n}{1 - 2^n}$. Then, the probability that we end with n dollars (we win $n - i$ dollars when we hit a boundary) is $1 - p_0 = 1 - \frac{2^i - 2^n}{1 - 2^n} = \frac{1 - 2^i}{1 - 2^n}$

(c) **(10 points)** Now suppose the initial number of dollars that each student has is a random number generated as follows: Every student is given n envelopes and each (independently) has 1 dollar with probability 1/2 and 0 dollars with probability 1/2. On day 0 the student opens all the envelopes to get their initial i dollars. Show that the probability of finishing with n dollars in this sequence of games is at least $\Omega(c^n)$ for some constant $c > 1/2$. (So while graduation is still exponentially unlikely, it is better than the probability of graduating on day 0.)

Proof. We wish to find P(finishing with n dollars and show that this is at least $\Omega(c^n)$ for some constant $c > 1/2$. Since the initial number of dollars that each student receives is based on n envelopes which each independently have a probability of $1/2$ of containing 1 dollar, we can express the probability that a student has an initial i dollars as:

$$\binom{n}{i}(\frac{1}{2})^i(\frac{1}{2})^{n-i}$$

$$\binom{n}{i}(\frac{1}{2})^n$$

Recall that we showed in part (c) that the probability of finishing with n dollars given i initial dollars is $\frac{1-2^i}{1-2^n}$. Since i is now a random variable, we apply the Law of Total Probability to find P(finishing with n dollars).

$$P(\text{finishing with } n \text{ dollars}) = \sum_{i=0}^{n} P(\text{end with } n\$ \mid \text{Start with } i\$) * P(\text{Start with } i\$)$$

$$P(\text{finishing with } n \text{ dollars}) = \sum_{i=0}^{n}\binom{n}{i}(\frac{1}{2})^n\frac{1-2^i}{1-2^n}$$

$$P(\text{finishing with } n \text{ dollars}) = \sum_{i=0}^{n}\binom{n}{i}(\frac{1}{2^n})\frac{2^i-1}{2^n-1}$$

Then, since $\frac{2^i-1}{2^n-1} \geq \frac{2^i}{2^n}$ because $i \leq n$, we have:

$$P(\text{finishing with } n \text{ dollars}) = \sum_{i=0}^{n}\binom{n}{i}(\frac{1}{2^n})\frac{2^i-1}{2^n-1} \geq \sum_{i=0}^{n}\binom{n}{i}(\frac{1}{2^n})\frac{2^i}{2^n}$$

$$P(\text{finishing with } n \text{ dollars}) \geq \frac{1}{2^{2n}}\sum_{i=0}^{n}\binom{n}{i}2^i$$

By the Binomial Theorem, we have that $\sum_{i=0}^{n}\binom{n}{i}2^i = (2+1)^n = 3^n$. Then,

$$P(\text{finishing with } n \text{ dollars}) \geq \frac{3^n}{4^n} = (\frac{3}{4})^n$$

Therefore, P(finishing with n dollars) $= \Omega((\frac{3}{4})^n)$, where $c = \frac{3}{4} > \frac{1}{2}$. □

(d) **(0 points, optional)** For every positive integer t, show that the probability that a student is neither bankrupt nor has graduated by the end of $10 \cdot t \cdot n$ days is at most $(23/30)^t$. [Hint: Suppose we change the rules so that students can play games even after they graduate or go bankrupt. What is the expected number of dollars they have after $10n$ days. What is the probability they have a positive number of dollars?]

(e) **5 points)** Show that the probability that a student graduates in $O(n^2)$ days is at least $\Omega(c^n)$ (for the same constant c as in Part (c)).

Proof. From part (d), we know that the probability that a student is neither bankrupt nor has graduated by the end of $10 \cdot t \cdot n$ days is at most $(23/30)^t$. Note that this probability is equivalent to the probability that a student *has not yet graduated* after $10 \cdot t \cdot n$ days. Therefore, let G denote the event that a student graduates and Y denote the event that a student has not yet graduated in $100n^2 = O(n^2)$ days. We wish to find $P(G \setminus D)$, as this is the probability that a student *graduates in* $O(n^2)$ *days*.

From Professor Sudan's Ed reply, we know that $P(G \setminus D)$ can be expressed as: $P(G \setminus D) \geq P(G) - P(D)$ From part (c), we know that $P(G) = \Omega(\frac{3}{4})^n$. From part (d), consider $t = 10n$. The probability that a student does not graduate in $10 * 10n * n = 100n^2 = O(n^2)$ days is then $O((\frac{23}{30})^{10n})$, by the formula. Therefore,

$$P(G \setminus D) \geq \Omega((\frac{3}{4})^n) - O((\frac{23}{30})^{10n})$$

Notice that $(\frac{23}{30})^{10n}) = 0.0702^n$ and $(\frac{3}{4})^n = 0.75^n$. Therefore, as n approaches infinity, $\Omega((\frac{3}{4})^n)^n \geq O((\frac{23}{30})^{10n})$ so our expression becomes:

$$P(G \setminus D) \geq \Omega((\frac{3}{4})^n)$$

Thus, we have thus shown that the probability that a student graduates in $O(n^2)$ days is at least $\Omega((\frac{3}{4})^n)$ where $\frac{3}{4}$ is the same constant c which we found in part (c).

□

(f) **(10 points)**: For some $c > 1/2$ give an expected $O(n^{124}c^{-n})$-time (therefore, expected $o(2^n)$-time!) algorithm for solving 3SAT. You may assume that the 3SAT formula is satisfiable.

Proof. Consider the following algorithm for solving 3SAT with n literals which can take true or false values. Note that this algorithm borrows heavily from the 3SAT algorithm given in lecture notes 18.

```
i. Repeat $T = c^{-n} = (\frac{3}{4})^{-n} = (\frac{4}{3})^n$ times
    A. Pick a random assignment of the $n$ literals to begin with. That is, assign
       each literal either true or false with $1/2$ probability.
    B. Repeat for $100n^2$ steps
        • If some clause is not yet satisfied, choose one of its variables at random
          and flip its value
        • Otherwise, terminate and output the current truth assignment
ii. RETURN UNSAT
```

To prove the correctness and runtime of our algorithm, we will show that our 3SAT algorithm is analogous to the Gossip Girl game which we have been dealing with in this problem. First, when we pick a random assignment at the beginning of the algorithm, we are essentially randomly picking a value 0 or 1 for each of the n literals. Notice that this is evidently analogous to the process of giving each student n envelopes which either contain 0 or 1 dollars. Then, for each of the $100n^2$ steps of the inner loop, in the worst case our algorithm flips a correct literal to the wrong value with $2/3$ probability and flips a wrong literal to the correct value with $1/3$ probability (note that wrong and correct are determined by the actual value of the specific literal in the satisfying assignment). Therefore, if we consider the algorithm in terms of how many literals at a given time step match the satisfying assignment, we increase the number of literals which match the satisfying assignment by 1 with probability $1/3$ and decrease the number of literals which match the satisfying assignment by 1 with probability $2/3$ at every step in the $100n^2$ steps. Notice that this is exactly analogous to the Gossip Girl game where we play $100n^2$ games on $100n^2$ days, which we win 1 dollar with probability $1/3$ and lose 1 dollar with probability $2/3$. Finally, notice that "winning" in 3SAT and the Gossip Girl game are analogous– Gossip Girl graduates upon reaching n dollars and 3SAT finds a satisfying assignment if n of its literals match the satisfying assignment.

Thus, the correctness of our algorithm follows from the correct 3SAT algorithm given in lecture notes 18. Since the only difference in our algorithm is we run for more steps, we conclude that if our algorithm returns an assignment that is not UNSAT, then the outputted assignment is indeed the satisfying assignment for the 3SAT problem. This is easily seen from the definition of the algorithm: we return the current truth assignment if and only if we have no clause that is not satisfied, which implies that all clauses are satisfied and we have a satisfying assignment. Next, we will prove that our algorithm is *expected* to find a satisfying assignment within the $T = (\frac{4}{3})^n$ loops of the outer loop. Note that the algorithm is essentially trying to find the first success, ie. when Gossip Girl has reached n dollars or when all n literals match the satisfying assignment. In addition, since the 3SAT algorithm is analogous to the Gossip Girl game, the probability that we find a satisfying assignment in one run of $100n^2$ steps is the same as the probability that Gossip Girl graduates in $100n^2 = O(n^2)$ days. As we showed in part (c), this is at least $(\frac{3}{4})^n$. Thus, we can view each run of $100n^2$ as having a "success" probability of $\Omega((\frac{3}{4})^n)$. Then, by the Expected Value of First Success, we have the expected number of trials for first success as $1/\Omega((\frac{3}{4})^n) = O((\frac{3}{4})^{-n}) = O((\frac{4}{3})^n)$. Since we set $T = (\frac{4}{3})^n$, then our algorithm is expected to find a "success" (the satisfying assignment) within the T loops of the outerloop.

The runtime of our algorithm is as such: we have at most $\binom{n}{3}$ clauses which contain 3 literals each (by definition of 3SAT). Therefore, we have $O(n^3)$ clauses to check in each of the $100n^2$ steps. Then, we also have $T = (\frac{4}{3})^n = (\frac{3}{4})^{-n}$ runs of the $100n^2$ steps. Hence, we have the runtime of our algorithm as $O(n^3 * 100n^2 * (\frac{3}{4})^{-n}) = O(n^5(\frac{3}{4})^{-n})$. Therefore, since we already proved that our algorithm is expected to solve 3SAT by the end of the T runs, our algorithm has expected time $O(n^5(\frac{3}{4})^{-n})$ to solve 3SAT which is $O(n^{124}c^{-n})$ where $c = \frac{3}{4} > \frac{1}{2}$ $\square$