Robinhood Casino

Version 1.4.4

December 15, 2019

Robinhood is a one of a kind casino service provider that eliminates the house, allowing players to play against each other. This document outlines the specifics of the business and is a reference guide for investors to research the offering. We assume readers have a basic understanding of blackjack and poker. We are publishing the press release now to allow our investors to examine the product offering from the perspective of future customers and assess the value it brings. The press release contains a section addressing questions frequently asked by customers once they learn about the product. The rest of the document discusses the market opportunity, the economics of the service and details of the token sale. You can find the most up-to-date version of this document at https://investors.robinhood.casino

Contents

1 Press Release

Introducing Robinhood. Putting trust back into gambling.

Until today gambling has been about players against the house, but with Robinhood this is no longer the case. Robinhood eliminates the house - the least trusted part of the equation - and allows players to directly play against each other. What Uber did to the taxi industry, Robinhood does to the casino industry.

It's very easy to play. First you login to the website and select the minimum bet you like to play with. Next, you are matched with a dealer via a live video feed. The dealer will start handing you cards and you'll be able to chat with other players, as if though you were physically present in the room! Feel like being the dealer? No problem! You can start your own table as a dealer and Robinhood will match you with other players on the website. To be a dealer, all you need is a webcam, deck of cards and shuffling machines, all of which is provided by Robinhood.

Anyone can be a card dealer. "I've always been a fan of blackjack and hosted games at home with my friends. With Robinhood, I turned my hobby into a paying job!" remarks Andreas, a Robinhood dealer from Brazil. Dealers don't have a manager at Robinhood and their performance is directly rated by the players. If a dealer's ranking falls below 4 out of 5, their license is automatically removed and they will no longer be able to participate. Thomas, a Robinhood player, said "because dealers know they are at the mercy of the players' rankings, they are being very customer obsessed. My dealer was even telling us jokes between each hand to entertain us."

Robinhood operates 24/7. Thanks to a global network of blackjack dealers, you can play whenever you feel like it. "It was 3 a.m. and I felt like playing some cards after having spent the night out. Instead of having to drive 2 hours to the closest casino, I logged into Robinhood and was matched with a dealer in Europe, all from the comfort of my bed!" remarked Sydney, a player from USA.

Players have the opportunity to challenge hands, if they feel there's been some wrong doing. If more than 50% of the players at the table do so, the money is locked until an auditor from Robinhood audits the video feed and makes a decision. Players can take comfort in knowing that they can expect fair play. Dealers who have problematic hands will be reviewed and banned from the platform if necessary. Additionally, the hand history of all dealers is publicly available for players to inspect and audit.

1.1 FAQ

How much does Robinhood cost? The platform is free of charge for players and will remain so indefinitely. Dealers, initially, also pay no fees and keep 100% of their winnings. However, Robinhood will eventually charge dealers a percentage of their winning. The timing of the fee, and its percentage will be decided by shareholders through decentralized autonomous organization (DAO) methodologies. Tips by players will not be subject to any fees.

What currency do players need? At start, Robinhood will support Ethereum but will expand to accept other crypto currencies such as BTC, Litecoin, Dash, etc. based on player feedback. Similar to a physical casino, players exchange Ether and Robinhood chips in the cashier section of the platform. The value of Robinhood chips is pegged to Ether so that players don't have to worry about price volatility while they are playing.

Why the name Robinhood? Robinhood is a well-known tale of the English folklore, a legendary heroic outlaw who took from the rich and gave to the poor. We feel we're applying the same transformation to the casino industry by distributing the profits to the players, instead of having it be limited to a few select operators. Players are empowered, no longer at a disadvantage facing giant casino houses which follow unfair practices.

What do dealers need to join? Dealers must have a computer with internet connection and need to purchase a starter pack priced at $195 (including shipping) which includes 6 decks of cards, a card shuffler, a card tray, a card scanner and a webcam. Using the card scanner, dealers swipe the cards they draw instead of manually entering the value and suite of the card in the system. This results in a better game experience since the game is not slowed down by manual data entry. It furthermore eliminates human error.

All items have clear Robinhood branding to help players validate that genuine Robinhood devices and cards are in use, preventing bad actors from using altered cards or devices to cheat.

Robinhood sells everything at cost and makes no profit on the starter pack to help dealers bootstrap their tables.

Why would someone become a dealer? Dealers can make anywhere from $34-$49 per hour for running tables with a $15 minimum player bet. For reference, this is 4-6X the minimum wage in United States of America. See subsection 4.2 for details. Aside, from the attractive income, dealers enjoy a great deal of autonomy, not having a boss or needing to report to anyone. They choose their own working hours and may opt to work from the comfort of their home.

How do dealers fund their tables? Dealers use Ether to buy chips, in the same manner as players, to fund their tables. They can cash out and convert their chips back to Ether whenever they please. Robinhood recommends a minimum and maximum table bet to dealers based on their balance and enforces a minimum to ensure solvency of the games.

How do players know dealers don't cheat? There are several mechanisms in place to bring players peace of mind that dealers are acting appropriately:

- Dealers are ranked by players on a scale of 1 (poor) to 5 (excellent). Any dealer with a score below 4 is banned from Robinhood. This mechanism empowers players to choose which dealers they like to see on Robinhood.
- Dealers purchase equipment (deck of cards, card shufflers, card scanners and camera) provided by Robinhood. Robinhood will send dealers branded equipment that it has internally tested and reviewed to ensure quality and fair play. Equipment will furthermore have tamper-proof seals to prevent alteration. Dealers are only allowed to use equipment sent to them by Robinhood and cannot use their own devices or cards.
- If, for any reason, players want to challenge a hand, they can do so conveniently from the platform. Money involved in challenged hands is blocked until a Robinhood auditor reviews the incident and resolves the case. Auditors will use player chat log, the dealer's video feed, player and dealer history to make their decision.
- Dealer hand history is available on Robinhood, and also published publicly, available for anyone to store and audit it at any time.
- Dealer statistics are continuously monitored to spot and audit outliers.
- Dealers have to comply with video feed rules that prevent use of any unapproved devices, or aids. Robinhood continuously validates dealer compliance and will ban non-compliant dealers.

2 Platform Features

Robinhood follows the Value Proposition Design methodology to ensure market fit for its platform. We identified 2 primary classes of users, players and dealers, and provide a summary of our findings for each user group below. For each user group, we identified the set of tasks they are going to perform, the negative outcomes associated with these tasks (pains) and ideal outcomes users seek (gains). We then tuned our product offering to minimize the pains, and amplify the gains for the users. While Robinhood will expand to support other games in the future, we place our emphasis on Blackjack for our initial milestone and aim to perfect our offering for Blackjack, before expanding to other card games such as Poker. As such, the contents of this section are solely focused on Blackjack.

2.1 Players

Robinhood's strategy is to continuously obsess over player satisfaction. Players are looking for a hassle-free, fun, and trustable gaming experience, and more than half prefer it on their mobile device[13]. The following features are launched in the initial milestone:

Depositing and withdrawing funds Players will be able to purchase chips in exchange for Ether. The value of the chips provided by Robinhood is pegged to the value of Ether such that users don't have to worry about volatility in the price of Ethereum. Players, at any time, can convert their chips back to Ether by going to the cashier section of the website. Players may use services such as ShapeShift to use fiat currencies to purchase chips.

Live chat with players and the dealer Players on a table will be able to chat with each other, and the dealer, in real-time. This will make the game more interactive and allows everyone to share their emotions as they play the game.

Watching tables Players will be able to watch tables without placing any bets. This allows players observe the gameplay of others and learn without spending any money.

Joining tables Players will be able to search for blackjack tables based on minimum and maximum bet amounts. Players can join tables using an internet browser from laptop and desktops, or on their mobiles devices (Android and iOS). Players can choose to play anonymously, or to open an account. Once players join a table, they see the live video-feed of the dealer, and the actions of all other players at the table.

Auditing dealers Players can view the dealers' profiles including their rankings and comments placed by other players. Players can additionally view an audit log of hands dealt by the dealer. Robinhood publishes real-time dealer hand data through a public event stream so that anyone can subscribe to, and store this data, providing full transparency on games. Players are free to either use Robinhood's archive to examine historical hand data, or to capture the data in real-time themselves if they don't want to trust Robinhood's archive.

Leaving tables and rating dealers Players are able to leave tables any time they desire (as long as they are not in the middle of a hand). Players are asked to rate their dealers and comment on their experience.

Challenging hands Players can challenge a hand if they feel there's been a mistake or if they sense suspicious activity. Upon clicking the challenge button, other players are asked whether they agree with the challenge. If more than 50% of the players agree, the funds associated with the hand will be frozen and the play continues to the next hand. Robinhood auditors will review the hand, the corresponding video and chat feeds, and make a decision as to whether there was any wrongdoing. The user experience will be similar to physical casinos where table managers are called to arbitrate situations.

2.2 Dealers

Dealers are equally key to Robinhood's success. Robinhood aims to provide a platform for dealers where they can earn income while enjoying their favorite game, blackjack. The following features are provided for dealers in the initial release:

Signing up as a dealer Dealers will be given a profile where they upload their picture and share information about themselves with the players. There are no restrictions on who can join, however, dealer performance is tracked via the ranking system and dealers who don't perform well are removed from Robinhood. Dealers don't have any managers to report to and may operate with whatever working hours they prefer. Upon sign up, dealers are required to purchase a starter pack, valued at $195 . This package includes a webcam, 6 decks of branded Robinhood cards, a card shuffler, a card tray, and a card scanner that allows quick scan of drawn cards. Robinhood makes no profit on the starter pack and provides it at cost to dealers to encourage adoption.

Withdrawing and depositing funds Dealers can exchange Ether for Robinhood chips and vice versa at any time. The value of the chips is pegged to Ether to address Ether price volatility. Ether will be supported initially and in later phases support for BTC, LiteCoin and other popular cryptocurrencies will be added to the platform.

Starting a table Dealers select the minimum and maximum bet amount they allow players to place. Based on this value, a minimum balance is calculated and enforced by Robinhood to ensure dealers are able to sustain the worst hits in case all players win in a hand. Once the table is started, the dealer's webcam is activated and players are automatically brought to the table.

Running hands Once the game starts, the dealers are instructed step-by-step on what actions to take at each turn. When players must act, the system tracks the player's action and displays it on the dealer's screen. The system will automatically track the value of the hands and will instruct the dealer on what steps to take. Dealers don't need to manually add card values, and can choose to strictly follow instructions provided by the platform.

Closing games The dealers may request the table to be closed at any time. Once requested, players are notified and upon completion of the ongoing hand, the table is closed. Players are automatically transferred to other tables. The dealer may then choose to withdraw the funds.

Interacting with players Dealers can interact with players through a live chat feed or through the video camera. Dealers are encouraged to provide a fun and entertaining experience for their players.

3 Market Opportunity

In 2017, land casinos in Europe, Australia and USA generated a revenue of $202 billion and employed 408,000 people, while the online sector in Europe brought in $22 billion and employed 33,000 people (see Table 2). Statistics for online operators are conservatively low, and hard to come by, as several operators, especially those based on cryptocurrencies, are still unregulated. Nonetheless, we observe tremendous opportunity for capturing revenue. Even a modest 0.1% revenue capture in the first year will result in a revenue of $202 million. Our research indicates a few key trends for growth that are discussed next.

3.1 Player mistrust

Lack of trust by players remains a key adoption blocker for new services. Reporters, journalists and whistle blowers regularly expose cheating schemes of online operators. Offline, players who do well are regularly banned by casinos without recourse. Even well-established operators such as PokerStars and Full Tilt Poker are consistently accused of misconduct and were even sued by the US government[12].

Cryptocurrency based operators aim to remove the trust busters by exposing their data publicly on the blockchain and playing into the provably fair trend[23]. Even though several operators have launched in this model, users still gravitate towards well known brands such as PokerStars. Exposing the track record on public blockchains is not sufficient because it only addresses a very small subset of tech-savvy players - with detailed understanding of blockchain technology - who can validate the data. The crux of the problem remains the mindset of being an underdog against the house. Robinhood believes changing the business model to remove the house from the equation, allowing any player to effectively become the house, will be a game changer and instrumental to gaining player trust. Furthermore, since dealers are incentivized to bring more players to the platform, this model results in free word-of-mouth marketing for the business.

3.2 Rise of Artificial Intelligence (AI)

While players are demanding convenience of playing from home, they are simultaneously worried about rise of bots in online casinos[7]. Robinhood addresses this by bringing the human element back to the game. Dealers will play out the games in real-time in front of the camera. Not only will this provide a fun and interactive experience akin to real casinos, it will also give our dealers the opportunity to spot bots and ban them from Robinhood. We believe simple day to day questions such as inquiring about one's day will be a good litmus test to identify bots. There's precedence

of success for live casino games, as demonstrated by operators such as bet365.com, casino.com, and leovegas.com giving Robinhood high confidence in the model.

3.3 User Experience Friction

Online gambling user experience (UX) suffers from the following pain points:

Deposits Credit card companies ban many gambling websites and players are often forced to purchase gift cards to fund their accounts. This means an extra trip to the closest gas station before the player can even start. Direct debit of bank accounts remains an unpopular funding option because players don't trust gambling operators, especially smaller ones, with their banking information. Cryptocurrencies are a major step forward in removing this friction for end users. Robinhood recognizes that cryptocurrencies result in friction for a subset of users in the short term, but anticipates that in the long run, cryptocurrencies will be adopted in mass and given their decentralized nature, are unbannable.

Withdrawals Neosafe, paysafecard, instadebit and instant eChecks are payment providers commonly used by gambling operators as withdrawal channels. All require either owning a credit card, or sharing personal banking information, and operate in a limited number of countries. Even large operators in the space, such as bet365, and PokerStars have no choice but to rely on physical bank wires, a very slow method, to pay a subset of their users. Robinhood will support instant withdrawals through Ether.

Gaming on the go Even though 51% of online gaming revenue comes from mobile[13], gambling with cryptocurrencies on the go remains uncharted territory. Browser extensions such as MetaMask only work on PC and laptops and leave a vacuum for mobile gaming. Robinhood will provide a first-class mobile gaming experience by integrating with Coinbase mobile wallet enabling users to play on the go.

4 Business Model

Robinhood operates as a service provider that matches players and dealers. The primary revenue source are fees levied on winnings of dealers. Robinhood starts with a promotional fee of 0% but will raise the fee to 30% as dealers adopt the platform. Robinhood's interest, therefore, lies in increasing volume of games played on the platform. Robinhood operates its business by the following tenets and believes these will lead to increased market share over time:

Operate with Full Transparency Robinhood will expose dealer hand history, provide mechanisms for users to control which dealers participate in the network, and make use of blockchain technology to bring transparency to financial operations of the platform.

Continously Lower Fees Robinhood will only charge dealers. Player winnings are the players to keep. Robinhood will continuously improve the efficiency of its operations and pass on the resulting savings to customers.

Provide Financial Freedom of Choice Players and dealers are able to use any cryptocurrency of their choice. Robinhood will never lock funds in proprietary tokens and will always allow players and dealers to convert their funds to their cryptocurrency of choice.

Bring Back the Human Touch Robinhood strives to create a life-like experience for players, providing an experience akin to visiting physical casinos.

4.1 Dealer Performance Management

To measure the performance of dealers on the platform, a rating score (out of 5) is assigned to each dealer. Robinhood will regularly conduct reviews of dealers with a score below 4 and may opt to remove dealers from the platform as a result. While it's critical to empower users to ban poorly performing dealers, it's of equal importance to ensure dealers are treated fairly. The review process enables Robinhood to keep this delicate balance.

To ensure players are in a balanced mindset while they write reviews, a 24 hour delay is incorporated from the time players leave the table until they are allowed to place their review. The platform notifies users to provide their feedback when the 24 hour window elapses. Reviews display the win/loss of the player for the session, as an additional measure, to assess truthfulness of the review.

4.2 Dealer Economics

The earnings of dealers can be calculated using the following equation:

$$E = \sum_{i=1}^{n} P_i * O * H \tag{1}$$

where:

E = dealer earnings
P_i = sum of bets wagered by player i
n = number of players at the table
O = dealer expected win ratio
H = number of hands played

The best strategy a player can follow (see appendix C) will yield a 0.859% dealer edge (see appendix D). It's unlikely that all players take the time to learn and play based on the perfect strategy, however. Many sources [15] [1] cite a typical house edge of 2% for blackjack. Land casinos play an average of 102 hands of blackjack per hour with 4 players at the table (see appendix E). Conservatively assuming that number of hands per hour in webcam settings are 40% of in-person rates, until users are fully educated on how to effectively use Robinhood, we can estimate the hourly expected earning of a dealer as follows:

$$E = \sum_{i=1}^{n} P_i * O * H = \sum_{i=1}^{4} P_i * 0.02 * 102 * 0.4 = 0.816 * \sum_{i=1}^{4} P_i = 0.816 * \$60 \tag{2}$$

The hourly expected income of dealers with a \$15 minimum bet (P_i) comes to \$49 at the beginning where dealers are exempt from fees. Even after applying a 30% fee to \$49 in later stages, the resulting hourly income of \$34.3 by far exceeds the minimum wage of all countries on the globe and is expected of a \$15 minimum bet table, the least expensive table setup in land casinos of North America. Robinhood believes this to be a very attractive figure for dealers to sign up on the platform.

4.3 Blockchain Use

Robinhood will be using the Ethereum blockchain to run its daily operations and also to manage its token sale. The Ethereum blockchain is an attractive choice because all parties can validate the logic behind smart contracts deployed to the network. Furthermore, all transactions can be publicly verified by all stakeholders which is a catalyst for investor and player trust. Lastly, due to the network's decentralized architecture, no one party can practically misbehave. These great features

of Ethereum come at the expense of state changes being very slow. This is necessary because all nodes in the network independently verify all transactions and state changes. As of March 2019, the Ethereum blockchain supports approximately 20 transactions per second. While there are plans to increase this rate to thousands per second [11], Robinhood does not build a dependency on this feature. To ensure a smooth gaming experience for players and dealers, Robinhood will run the games off-chain, using its own proprietary backend, scaling to milliseconds latency for hand actions. Each action is published through event streams that any interested party can listen to, as a mechanism to enable third parties to validate transactions. Blockchain use will be limited to withdrawals, deposits, and actions related to investors such as voting on Robinhood's dealer fee structure.

4.4 Revenue Projections

This section presents revenue projections for the 1st year of the business. We present two scenarios: a) Blackjack, playable on desktop, and b) Blackjack and Poker, playable on desktop and mobile devices. A complete sale of tokens will enable Robinhood to pursue b), while an incomplete sale of tokens (softcap), will force Robinhood to only pursue a).

Robinhood projects a revenue of $538,674.48 for scenario a, and $1,848,004.64 for scenario b, for the first 12 months after launch of the product. The rest of this section details how these figures are obtained. Note these figures represent only 0.000024 and 0.000084 of the total $22 billion revenue captured by the online gambling sector (see section 3). This highlights the massive opportunity for growth available to Robinhood.

4.4.1 Blackjack

The revenue from Blackjack can be represented as:

$$BR = \sum pb * dp * f \tag{3}$$

where:

BR = expected revenue from Blackjack
dp = expected card dealer profit from player bets
f = fee imposed by Robinhood on dealer profit
pb = bet placed by a player

Robinhood will charge a fee of 30% to dealers. If players take the best possible choice at every turn of the game, the card dealer is expected to have a win rate of 0.8% (appendix D). Several websites quote this figure to be close to 2% ([15] [1]) and some claim 5%-10%. This is because players don't play the best strategy (appendix C) and are influenced by environmental factors and emotions. It's atypical for players to quit when they are ahead. We assume dealer's winrate to be 2%, therefore BR can be rewritten as $BR = \sum pb * 0.02 * 0.3$.

$\sum pb$ is a function of how many players are on the platform and how many hands they play. We can expand $\sum pb$ as follows:

$$\sum pb = pc * ph * ab \tag{4}$$

where:

pc = how many players are active on the platform in a year
ph = how many hands a player plays in the course of a year
ab = average bet size of each hand

To estimate *ab*, *conservatively*, we assume Robinhood will only run tables with smallest minimum bet amount of $15 and maximum bet amount of $45. These are the minimums found at casinos across North America. Using these figures, we estimate the average hand to be $ab = average(15, 45) = 30$.

Each active player spends 4.065 hours per week on the game (see appendix H). Casinos play an average of 102 hands of blackjack per hour (see appendix E). We expect games on Robinhood to be slower due to the overhead of webcams, and remote players, so we add a 50% penalty to 102. Therefore, we expect $\sum pb = pc * ph * ab = pc * 4.065 * 102 * 0.5 * 52 * \$30 = \$323411.40 * pc$

This figure represents the bets placed by an active player over the course of a year. Please note this is different than how much capital a player has. For example, a player with a bank of $10, might bet their $10, one hundred times. Such player would contribute $1,000 to the above figure.

Replacing $\sum pb$ in equation 3, we obtain:

$$BR = \$323411.40 * pc * 0.02 * 0.3 = 1940.47 * pc \tag{5}$$

This equation represents the expected generated revenue by an active player over the course of a year.

4.4.2 Poker

The revenue from Poker can be represented as:

$$PR = \sum min(pot * r, mr) * f \tag{6}$$

where:

PR = expected revenue from Poker
r = table rake
mr = limit on the rake
f = fee imposed by Robinhood on dealer profit
pot = total money placed by players in the pot

Robinhood will charge a fee of 30% on dealer profit and set the rake of the tables to 10%, similar to what players find in casinos. The rake typically has a maximum amount, regardless of how much money is in the pot. This is normally set to $5 in casinos and Robinhood will replicate the same setup. PR therefore can be rewritten as $PR = \sum min(pot * 0.1, 5) * 0.3$. Given the average pot size of $27 for $1/$2 stakes game with 6 players (see appendix G), this can be reduced to $PR = \$2.7 * 0.3 * h$, where h is the count of hands played on tables with 6 players.

To estimate number of hands played at each table, we rely on how much time players spend on the platform and how many hands are played during that time. Players spend 4.065 hours per week on the game (see appendix H). In PokerStars, tables run 97.19 hands per hour (see appendix G). We expect a lower hand count per hour at Robinhood since dealers are physically handing cards, in contrast to PokerStars where this is done electronically. To account for this fact, we apply a 50% reduction to the hourly rate, arriving at $97.19 * 0.5 = 48.60$. Each table is then expected to yield $48.60 * 4.065 = 197.56$ hands per week, and $197.56 * 52 = 10273.12$ per year. It takes 6 players to create a table, therefore, each player is expected to contribute to $10273.12/6 = 1712.19$ hands per year. Substituting for h, we obtain:

$$PR = \$2.7 * 0.3 * 1712.19 * pc = 1386.87 * pc \tag{7}$$

where:

pc = number of active players in a given year

4.4.3 Player Acquisition

Equations 3 and 7 express the revenue in terms of how many players are active on the platform. Robinhood will primarily acquire players through online marketing. This can be projected by:

$$pc = av * ctr * sr \tag{8}$$

where:

pc = how many players will be active on the platform
av = how many ads are viewed by players
ctr = click-through rate of ads shown to players
sr = percentage of players signing up after having clicked on the ad

Average click-through rate for gambling ads on YouTube is 0.25%[22]. The average click-through rate for Google AdWords in the Dating & Personal category is 1.96%[24]. Facebook's average click-through rate across all categories is 0.9%[25]. Finally, Instagram's average click-through rate across all categories is 0.52%[17]. As such, we set $ctr = average(0.0025, 0.0196, 0.009, 0.0052) = 0.009075$.

To estimate sr, we can rely on bounce-rate, a metric that defines how many visitors of a website only look at one page and leave. The average bounce-rate is 49%[16], out of the 51% who visit more pages, we expect to capture 3-6%. We believe these to be reasonable assumptions since the player has landed on the website after having clicked on an ad that interested them. We'll assume $sr = 0.51 * 0.03 = 0.0153$ if Robinhood does not support mobile devices (scenario a), and $sr = 0.51 * 0.06 = 0.0306$ in case support for mobile devices is also launched (scenario b). Equation 8 can be rewritten as:

$$pc = av * ctr * sr = \begin{cases} av * 0.009075 * 0.0153 = 0.0001388 * av, & \text{for scenario a)} \\ av * 0.009075 * 0.0306 = 0.0002777 * av, & \text{for scenario b)} \end{cases} \tag{9}$$

With the above, TR, total revenue, can be modeled as following for scenario b:

$$TR = BR + PR \tag{10}$$
$$TR = 1940.47 * pc_{blackjack} + PR \tag{11}$$
$$TR = 1940.47 * pc_{blackjack} + 1386.87 * pc_{poker} \tag{12}$$
$$TR = 1940.47 * 0.0002777 * av_{blackjack} + 1386.87 * 0.0002777 * av_{poker} \tag{13}$$
$$TR = 0.53886852 * av_{blackjack} + 0.38513380 * av_{poker} \tag{14}$$

Targetting 2,000,000 ad views for both Blackjack and Poker, Robinhood will generate a total revenue of $1,848,004.64.

TR, for scenario a can be modeled as:

$$TR = BR \tag{15}$$
$$TR = 1940.47 * pc_{blackjack} \tag{16}$$
$$TR = 1940.47 * 0.0001388 * av_{blackjack} \tag{17}$$
$$TR = 0.26933724 * av_{blackjack} \tag{18}$$

Targetting 2,000,000 ad views, Robinhood can generate a total revenue of $538,674.48.

Acquisition Cost Cost per thousand impressions (CPM) at Instagram is $7.91 [17], and at Google is $2.8 [3]. Similarly, cost per click (CPC) averages to $1.41 at Instagram [17], and $0.75 at Google [3]. To estimate aquisition cost, we consider an average $CPM = (7.91 + 2.8)/2 = \$5.35$ and average $CPC = (1.41 + 0.75)/2 = \$1.08$.

Targetting 2,000,000 views (scenario a) and 4,000,000 views (scenario b) will cost $10,700 and $21,400 respectively. From equation 8, we know the total number of clicks is given by $av * ctr$ which is $0.009075 * av$. This results in 18,150 clicks in scenario a and 36,300 clicks in scenario b. The cost of these clicks, given CPC, will be $19,602.00 and $39,204.00 respectively.

The per player acquisition cost is given by:

$$pac = as/pc \tag{19}$$

where:

pac = player aquisition cost
as = total ad spent in $
pc = count of players

Plugging in equation 8, we obtain

$$pac = as/pc = as/(av * ctr * sr) \tag{20}$$

For scenario a)

$$as = \$10{,}700 + \$19{,}602.00 = \$30{,}302 \tag{21}$$
$$pc = 2{,}000{,}000 * 0.009075 * 0.0153 = 277.695 \tag{22}$$
$$pac = as/pc = \$109.12 \tag{23}$$

For scenario b)

$$as = \$21{,}400 + \$39{,}204.00 = \$60{,}604 \tag{24}$$
$$pc = 4{,}000{,}000 * 0.009075 * 0.0306 = 1110.78 \tag{25}$$
$$pac = as/pc = \$54.56 \tag{26}$$

5 Investment Prospectus

2717739 ONTARIO INC. ("Robinhood", "Corporation") is offering 5,000,000 common shares in the captial of Corporation ("Common Shares") at a price range of $0.14879930 to $0.19839907. The Common Shares qualified hereunder are referred to herein as the "Offered Shares".

Offered Shares are sold in 5 rounds. Each round has a limited capacity, and a pre-determined share price, as described in the table below. Once capacity is exhausted at a certain round, the sale proceeds to the next. Early bird investors are able to acquire tokens at a significantly cheaper price compared to those who invest later. The following table summarizes the token supply and price of each round:

	Token Price	Discount	Token Supply
Round 1	$0.14879930	25%	1,000,000
Round 2	$0.15871926	20%	1,000,000
Round 3	$0.16863921	15%	1,000,000
Round 4	$0.17855916	10%	1,000,000
Round 5	$0.19839907	0%	1,000,000
Average Token Price	$0.17062320		
Total Token Quantity			5,000,000

Table 1: Token Sale Breakdown

Unlike traditional offerings, there are no Underwriters at play. Investors directly acquire shares by sending Ether to Robinhood's Smart Contract on the Ethereum blockchain at address 0x218E8aEFF49340E4f6d34FE255d17B589dacEc4F. The smart contract will automatically issue investors shares according to how much money they've sent and the current sale round. There are no humans involved in this process.

Investors are able to inspect the source code of the smart contract and validate its behavior on their own. Furthermore, the Offered Shares are also represented as a smart contract on the Ethereum blockchain. Investors can inspect the source code at EtherScan. The token smart contract allows investors to check their balance by invoking the "balanceOf" function, free of charge.

If the softcap of $368,883.00 (2291.20 ETH[1]) is not reached, all money will be returned to investors. In case the softcap hits, shares will be available for trading on exchanges 9 months after the completion of the sale. This is designed to prevent manipulation of the token price by traders.

The tokens are issued automatically on-demand by the crowdsale smart contract as investors deposit funds (see page 47). Once the sale is completed, the tokens will be doubled and the resulting half will be reserved for founders, development team, advisors, and legal fees. The crowdsale smart contract imposes a 14 months vesting period to tokens issued for aforementioned stakeholders to ensure they stay invested in the long term success of the company. This mechanism also protects investors against insiders prematurely cashing out their tokens.

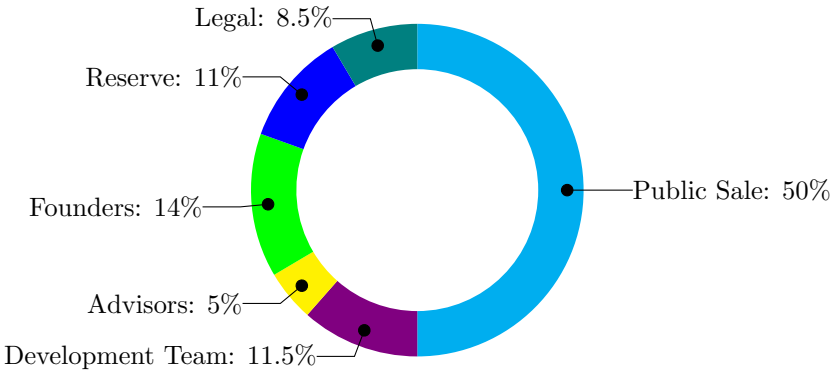

Figure 1: Distribution of Tokens

[1] Assuming 1ETH = $161

Investors should rely only on the information contained in or incorporated by reference in this prospectus. The Corporation has not authorized anyone to provide investors with different information. The Corporation is not offering the Offered Shares in any jurisdiction in which the Offering is not permitted. Investors should not assume that the information contained in this prospectus is accurate as of any date other than the date on the front of this prospectus or the date of such documents incorporated by reference herein, as the case may be. Subject to the Corporation's obligations under applicable securities laws, the information contained in this prospectus or any documents incorporated by reference herein is accurate only as of the date of this prospectus or the date of such documents incorporated by reference herein, regardless of the time of delivery of this prospectus or of any sale of the Offered Shares. The Corporation's business, financial condition, results of operations and prospects may have changed since those dates.

An investment in the Offered Shares involves a high degree of risk. Certain risk factors in respect of the Corporation and an investment in the Offered Shares should be carefully reviewed and evaluated by investors. See "Risk Factors" and "Forward Looking Statements" herein.

5.1 Business of the Corporation

Robinhood's strategy is to empower players by eliminating gambling and casino houses. Popular games such as Poker, Blackjack, Baccarat, Roulette, etc. need card dealers who are employed & managed by casino establishments. Establishments are trust-busters since there's no visibility into their operations and conduct. Robinhood enables any player to assume the role of the card dealer and removes the need for casino establishments. Players control and self-govern the system by removing poorly performing card dealers, while Robinhood provides necessary tooling and support.

Robinhood charges card dealers a commission on their winnings. Robinhood's incentive, therefore, lies in increasing total hands played through the system. The income is independent of player or dealer winning odds, since losses by either party don't incur any additional costs for Robinhood. This allows Robinhood to remain unbiased. To increase platform adoption early on, Robinhood will start with a commission of 0, and increase it gradually over time. Shareholders will vote on fee increases and their timings.

Robinhood has created a game experience similar to that of casinos. A live video feed of card dealers is streamed to all players on the table. Players can talk to the dealer and other players just like they would in a casino setting.

To drive adoption, Robinhood will focus on addressing the following user pain points:

- Friction of online payments for gambling websites. Players often are forced to buy Visa or MasterCard gift cards to participate in online gambling.
- Bots. Bots competing against humans in online Poker websites are on the rise and common cause of player complaint.
- Unfair practices by the house. Operators tamper with random number generators, and mine player data to gain an unfair advantage.

5.2 Description of Securities

The Corporation is authorized to issue an unlimited number of Common Shares and an unlimited number of Preferred Shares, issuable in series. As at the date hereof, there were 0 Common Shares issued and outstanding and no Preferred Shares outstanding. The following is a summary of the rights, privileges, restrictions and conditions attaching to the Common Shares.

5.2.1 Common Shares

Shareholders of Robinhood are entitled to one vote per Common Share at the meetings of shareholders, dividends, if, as and when declared by the Board and, upon liquidation, to share equally in the assets of Robinhood that are distributable to shareholders.

5.3 Prior Sales

No prior sale has been conducted, however, a total of 5,000,000 shares will be issued for founders, development team, advisors, legal fees, and bonus programs for the community. The crowdsale smart contract imposes a 14 months vesting period on shares of aforementioned stakeholders to ensure they stay invested in the long term success of the company. This mechanism also protects investors against insiders prematurely cashing out their shares.

5.4 Token Utility

5.4.1 Voting Rights

The tokens will give investors the right to vote on key business decisions. Investors will have a say proportional to the amount of their investment, as long as they own a minimum of 1,200 shares. The voting is modeled as a Decentralized Autonomous Organization (DAO) smart contract on the Ethereum blockchain. As an example, one of the key business decisions that will be put up for voting is the fee charged to card dealers. Robinhood will start with a fee of 0% and increase it over time as more dealers adopt the platform. Each fee raise, and the target value will be put up for voting through the DAO. Investors can find this smart contract at page 57 and inspect its source code and tests. To cast a vote, investors invoke the smart contract's CastVote method using their wallet and provide their preferred choice as an argument to the function call.

5.4.2 Investment Returns

Dividends At the end of every quarter, Robinhood will issue earnings statements similar to public companies listed on the NASDAQ exchange. Net profit for the quarter will be shared with token holders via dividends. The amount of dividend paid is controlled through the DAO voting mechanism presented above, but will start at 50%. Expected dividend income per share per year can be modeled as follows:

$$EI = \frac{r * pm * dp}{10{,}000{,}000} \tag{27}$$

where:

EI = expected passive income from investment
r = revenue generated by Robinhood
pm = profit margin of Robinhood on revenue
dp = percentage of net income paid out as dividend

Across 92 firms in the US, the gross margin for software companies in the entertainment sector is 65.91%, so we set $pm = 0.6591$[8]. Dividend paid to investors will start at 50% and can be controlled through the DAO mechanism, leading to $dp = 0.5$. Plugging the values into 27, we obtain:

$$EI = \frac{r * 0.6591 * 0.5}{10{,}000{,}000} = \frac{r * 0.3296}{10{,}000{,}000} \tag{28}$$

Based on revenue projections presented in 4.4.1, we obtain EI = \$0.017755 for scenario a and EI = \$0.060910 for scenario b. Given an average share price of 0.17062320, these represent a yearly return of 10.41% and 35.70% respectively.

5.5 Participation

Robinhood follows Know Your Customer (KYC) procedures and asks investors to provide their identity information to participate in the sale. Investors can visit `https://investors.robinhood.casino` to obtain an investor registration number. Even though Robinhood provides guidance on regulations to investors, participation is not restricted and it's left to investors to decide whether the rules and regulations of their country of residence allows them to participate in the sale. Robinhood will use the registration information provided by investors to communicate business matters, quarterly income statements and votes through the DAO.

RHC tokens are considered securities according to the USA Securities and Exchange Commission (SEC). Robinhood has filed a Form D with the SEC to enable US investors to participate. US investors can validate Robinhood's Regulation D compliance by visiting SEC's EDGAR website and searching by company name of Robinhood Casino.

5.5.1 Bonus Programs

Robinhood runs the following bonus programs through which community members can earn tokens, in exchange for an activity that raises awareness about Robinhood. Robinhood encourages the community to stay objective and provide constructive feedback so that it can iteratively improve Robinhood with the community's help. To claim your tokens, please email your investor registration number and all relevant information to contact+bonus@robinhood.casino.

Joining Telegram Group Joining the Robinhood's telegram group earns investors 5 tokens. Please include your Telegram username in your email.

Blog Post Writing a blog post reviewing Robinhood, providing feedback on the business plan and helping the community and Robinhood improve will earn you 235 tokens. Please include a link to your blog post in your email.

YouTube Review Doing a YouTube review of the crowdsale will earn you 590 tokens. Please include a link to your video in your email.

Sharing in Social Media Tweeting a link to the website, sharing it on Facebook profile or Instagram will earn users 5 tokens. Please include a link to your tweet/profile in your email.

Bounty Program If you bring any investor who purchases at least 6,000 shares, Robinhood will pay you a commission of 5%.

5.6 Use of Proceeds

The sale's softcap is valued at $368,883.00 (2291.20 ETH[2]). Funds will be deployed to expedite launch of Blackjack on the main Ethereum network (M8 milesone of the Roadmap).

Initially, more players will join the platform, compared to dealers. To ensure player demand is appropriately met by card dealers, Robinhood will boostrap the system with 5 dealers, who will be hired as employees. Robinhood will fund the tables for said dealers, and pay them a salary of $11.8 per hour (minimum wage in Canada). We anticipate to have sufficient organic dealer participation after 3 months and will retire Robinhood's own dealers.

[2] Assuming 1 ETH = $161 USD

The expenses for the first year follow:

Staff		$272,000.00	73.74%
Software Developer 1	$94,000		
Software Developer 2	$94,000		
UX Designer	$84,000		
Marketing		$34,802.00	9.43%
Ad Display (see 4.4.3)	$30,302		
Video Ad - Actors and Voice Actors	$4000		
Video Ad - Equipment Rental	$500		
Dealers (5 Tables, 3 months)		$35,070.00	9.51%
Table Bank [3]	$6750.00 (1350*5)		
Dealer Salary [4]	$28,320.00		
Office & Equipment		$20,450	5.54%
Rent [5]	$18,000.00		
Misc.	$500		
Dealer Equipment [6]	$1950		
Cloud Services		$6561	1.78%
Dacast Video Streaming - 5 months [7]	$1950		
AWS Services - 12 months	$4611		
Total		**$**368,883.00	**100%**

To tackle Android and iOS support from the get-go, 2 additional developers ($188,000) are necessary. To tackle Poker, in addition to Blackjack, two more developers are necessary ($188,000). To bootstrap poker tables with 5 dealers, for 3 months, a salary of $28,320.00[8] is necessary. Finally, adding a customer service representative to ensure prompt responses to customer asks will require an additional $45,000. Lastly, to market the poker offering, an additional budget of $30,302.00 (see 4.4.3) is necessary. Cloud service costs will increase by $4611. In total, extra costs to tackle Poker and mobile support come to $484,233.

The softcap ($368,883.00) will fund the expenditures for first year, without Poker, and without mobile support. The hardcap ($853,116) allows us to tackle all problems at once.

[3] Dealer chance of bust is 28% with 6 decks of cards, standing on 17. With 6 players at the table, each with a max bet of $45, a full bust costs $270. Chance of dealer busting 5 times in a row is 0.04%. To last this extreme outcome, a dealer needs to have $1350.

[4] $11.8 USD minimum wage, 40 hours a week, full-time salary, 5 dealers for 3 months

[5] $500 per desk per month, 3 desks for 12 months

[6] Inventory of dealer equipment for 10 dealers (e.g. deck of cards, card shufflers, card shoes, webcams, etc.)

[7] 3rd party video streaming provider. Services are not necessary for the first 7 months. $390 per month. See https://www.dacast.com/live-streaming-pricing-plans/

[8] Poker tables require no capital to run, therefore the cost is only the salary of the dealers

While the Corporation presently intends to use the net proceeds of the Offering as stated above, there may be circumstances that are not known at this time where a reallocation of the net proceeds may be advisable for business reasons that management believes are in the Corporation's best interests.

Until required for the Corporation's purposes, the proceeds will only be invested in securities of, or those guaranteed by, the Government of Canada or any Province thereof or the Government of the United States of America, in certificates of deposit or interest-bearing accounts of Canadian chartered banks, trust companies, credit unions or the Ontario Treasury Branches.

5.7 Risk Factors

Prospective investors should carefully consider all of the information set out in this prospectus and in the whitepaper and the risks attaching to an investment in the Corporation, including, in particular, the factors set out in this section, before making any investment decision. Investors should consider carefully whether an investment in the Offered Shares is suitable for them in light of the information in this prospectus and in the whitepaper. Such information does not purport to be an exhaustive list. If any of the identified risks were to materialize, the Corporation's business, financial position, results and/or future operations may be materially affected. Additional risks and uncertainties not presently known to the Corporation, or which the Corporation currently deems immaterial, may also have a material adverse effect on the Corporation's business and prospects.

There can be no certainty that the Corporation will be able to implement successfully the strategy set out in this prospectus and in the documents incorporated by reference herein. No representation is or can be made as to the future performance of the Corporation and there can be no assurance that the Corporation will achieve its objectives.

Investors should consult their own professional advisors to assess the tax, legal and other aspects of an investment in the Offered Shares.

5.7.1 Ethereum Platform

The Corporation relies on the Ethereum platform for its crowdsale and many of its mission critical features. The Corporation does not have control over reliability of the Ethereum platform, its availability, or cyber-security threats it faces. Problems in the Ethereum platform can materially impact the financial well-being of the Corporation.

5.7.2 Regulatory Uncertainty

Many jurisdictions do not yet have clear guidance and regulatory frameworks around Security Token Offerings (STOs). The Corporation is adhering to all policies to the best of its capabilities, however, changes to regulations, especially to those applicable to North America pose an operational risk. Regulation changes may negatively impact the performance of the Corporation, and thereby the value for shareholders, and occur in a manner completely out of control of the Corporation.

5.7.3 Use of Proceeds of the Offering

The Corporation intends to use the net proceeds of the Offering as set out under "Use of Proceeds" (section 5.6) in this whitepaper. However, these allocations are based on the current expectations of management of the Corporation and there may be circumstances that are not known at this time where a reallocation of the net proceeds of the Offering may be advisable for business reasons that management believes are in the Corporation's best interests.

5.8 Eligibility for Investment

In the opinion of counselors of the Corporation, the Offered Shares will be a "qualified investment" for trusts governed by a registered retirement savings plan ("RRSP"), a registered education savings plan, a registered retirement income fund ("RRIF"), a deferred profit sharing plan, a registered disability savings plan and a tax-free savings account ("TFSA") (collectively "Exempt Plans").

Notwithstanding the foregoing, the annuitant of a RRSP or a RRIF or the holder of a TFSA, as the case may be, will be subject to a penalty tax if the Offered Shares held in a RRSP, RRIF or TFSA are a "prohibited investment" and not "excluded property" for the purposes of the Tax Act. Offered Shares will generally be a "prohibited investment" if the annuitant or the holder, as the case may be, does not deal at arm's length with the Corporation for the purposes of the Tax Act or has a "significant interest" (as defined in the Tax Act) in the Corporation. Prospective purchasers who intend to hold Offered Shares in an Exempt Plan should consult their own tax advisors with respect to their individual circumstances.

The Corporation has filed a Form D filing with the Securities and Exchange Commission (SEC) of the United States. Investors from USA may acquire Offered Shares, according to provisions and regulations laid out by the SEC to participate in foreign offerings under Reg 506 (c).

The Corporation is not offering the Offered Shares in any jurisdiction in which the Offering is not permitted. Unless stated otherwise, purchasers bear the responsibility to validate their participation in the sale is non-problematic with applicable laws in their jurisdiction.

5.9 Purchaser's Statutory Rights

Securities legislation in certain of the provinces of Canada provides purchasers with the right to withdraw from an agreement to purchase securities. This right may be exercised within two business days after receipt or deemed receipt of a prospectus and any amendment. In several of the provinces, the securities legislation further provides a purchaser with remedies for rescission or, in some jurisdictions, revisions of the price or damages if the prospectus and any amendment contains a misrepresentation or is not delivered to the purchaser, provided that the remedies for rescission, revision of the price or damages are exercised by the purchaser within the time limit prescribed by the securities legislation of the purchaser's province. The purchaser should refer to any applicable provisions of the securities legislation of the purchaser's province for the particulars of these rights or consult with a legal adviser.

5.10 Forward Looking Statements

This whitepaper contains forward-looking statements concerning the business, operations and financial performance and condition of Robinhood. All statements other than statements of historical fact contained in this whitepaper are forward-looking statements, including, without limitation, statements regarding the future financial position, business strategy, proposed acquisitions, budgets, litigation, projected costs and plans and objectives of or involving Robinhood. Often, but not always, forward-looking statements can be identified by the use of words such as "plans", "believes", "expects", "will", "intends", "projects", "anticipates", "estimates", "forecasts", "budgets", "continuous" or similar words or the negative thereof.

Forward-looking statements involve known and unknown risks, uncertainties and other factors that may cause actual results or events to differ materially from those anticipated in such forward-looking statements. The forward-looking statements in this whitepaper reflect the current expectations,

assumptions or beliefs of Robinhood based on information currently available to it and on management's experience and expertise. Examples of such statements include: (A) the intention to complete the Offering; and (B) the intention to grow the business and operations of Robinhood. Many factors could cause the actual results, performance or achievements to be materially different from any future results, performance or achievements that may be expressed or implied by such forward-looking statements, including, without limitation, those listed in this whitepaper.

Should one or more of these risks or uncertainties materialize, or should assumptions underlying the forward-looking statements prove incorrect, actual results, performance or achievements may vary materially from those expressed or implied by the forward-looking statements contained in this whitepaper. These factors should be considered carefully and prospective investors should not place undue reliance on the forward-looking statements.

Although the forward-looking statements contained in this whitepaper are based upon what management currently believes to be reasonable assumptions, Robinhood cannot assure prospective investors that actual results, performance or achievements will be consistent with these forward-looking statements. Robinhood assumes no responsibility to update forward looking statements, other than as may be required by applicable securities laws.

6 Roadmap

Robinhood will pursue an aggressive roadmap with an iterative development philosophy where feedback is sought and incorporated early and in small increments. The product will be internally tested, followed by a beta launch on the Ropsten network to get external user feedback. The following table outlines the milestones (see appendix F for detailed acceptance criteria for each milestone).

M0	ERC20 token, crowdsale and DAO contracts
M1	Investor website with KYC registration
M2	UX Study
M3	Prototype Demo
M4	STO Launch
M5	Beta Launch on Ropsten Network
M6	Early Customer Feedback
M7	Code Complete
M8	Launch on main Ethereum Network
M9	Listing on Exchange
M10	Poker Cash Games
M11	Android and iOS Launch
M12	Poker Tournaments
M13	Baccarat

7 Questions & Feedback

Robinhood encourages investors to send their questions and feedback to contact+investors@robinhood.casino. This document is regularly updated with answers to common questions and revised according to feedback provided by participants.

Appendix A Industry Revenue and Employee Size

Year	Country	Revenue (USD millions)	Employees	Industry	Source
2017	USA	40,280	361,000	Commercial Land Casinos	[4]
2011	UK	4,550	38,800	Retail and Remote Betting	[9]
2015	UK	2,470	34,000	Gaming machines and arcades	[18]
2017	Germany	770.5	4,836	Land casinos	[6]
2016	France	2,464	14,500	Land casinos	[5]
2016	Poland	142.2	2,000	Land casinos	[5]
2016	Spain	382	5,000	Land casinos	[5]
2016	Italy	362	1,722	Land casinos	[5]
2016	Austria	367	1,427	Land casinos	[5]
2016	Sweden	138	927	Land casinos	[5]
2016	Portugal	332	2,100	Land casinos	[5]
2016	Finland	32	132	Land casinos	[5]
2016	Switzerland	7,470	2,024	Land casinos	[5]
2016	Slovenia	178	1,788	Land casinos	[5]
2016	Czech Republic	252	3,952	Land casinos	[5]
2016	Denmark	68	300	Land casinos	[5]
2016	Greece	301	3,710	Land casinos	[5]
2016	Netherlands	650	2,519	Land casinos	[5]
2016	Luxemburg	45	200	Land casinos	[5]
2016	Slovakia	31	570	Land casinos	[5]
2017	Australia	148,111	N/A	All	[19]
2017	Europe	21,952	33,000	Online gambling	[2]
2013	China	12,880	N/A	Online gambling	[20]

Table 2: Gambling Industry Size

Appendix B Blackjack Rules

Blackjack on the platform will be played with the following rules in place:

- Dealer hits on soft 17. If the dealer receives a 17 with an Ace, the dealer will hit
- Players are allowed to only double their bets on 9, 10 and 11
- Players are allowed to double soft hands. For example, a double of a 9, composed of at 8 and an Ace is allowed
- Players are allowed to resplit all hands, except for when they have 2 Aces
- Players are not allowed to double after a split

- Dealers will use 6 decks of cards
- Blackjack pays 3:2

Appendix C Blackjack Player's Best Strategy

The following table demonstrates actions a player can take in every given possible situation, to maximize odds of winning the hand.

Player Cards	Dealer Upcard									
	2	3	4	5	6	7	8	9	10	A
5	h	h	h	h	h	h	h	h	h	h
6	h	h	h	h	h	h	h	h	h	h
7	h	h	h	h	h	h	h	h	h	h
8	h	h	h	h	h	h	h	h	h	h
9	h	dh	dh	dh	dh	h	h	h	h	h
10	dh	dh	dh	dh	dh	dh	dh	dh	h	h
11	dh	dh	dh	dh	dh	dh	dh	dh	dh	dh
12	h	h	s	s	s	h	h	h	h	h
13	s	s	s	s	s	h	h	h	h	h
14	s	s	s	s	s	h	h	h	h	h
15	s	s	s	s	s	h	h	h	h	h
16	s	s	s	s	s	h	h	h	h	h
17	s	s	s	s	s	s	s	s	s	s
18	s	s	s	s	s	s	s	s	s	s
19	s	s	s	s	s	s	s	s	s	s
A,2	h	h	h	h	h	h	h	h	h	h
A,3	h	h	h	h	h	h	h	h	h	h
A,4	h	h	h	h	h	h	h	h	h	h
A,5	h	h	h	h	h	h	h	h	h	h
A,6	h	h	h	h	h	h	h	h	h	h
A,7	s	s	s	s	s	s	s	h	h	h
A,8	s	s	s	s	s	s	s	s	s	s
A,9	s	s	s	s	s	s	s	s	s	s
A,A	ph	ph	ph	ph	ph	ph	ph	ph	ph	ph
2,2	h	h	ph	ph	ph	ph	h	h	h	h
3,3	h	h	ph	ph	ph	ph	h	h	h	h
4,4	h	h	h	h	h	h	h	h	h	h
5,5	dh	dh	dh	dh	dh	dh	dh	dh	h	h
6,6	h	ph	ps	ps	ps	h	h	h	h	h
7,7	ps	ps	ps	ps	ps	ph	h	h	h	h
8,8	ps	ps	ps	ps	ps	ph	ph	ph	ph	ph
9,9	ps	ps	ps	ps	ps	s	ps	ps	s	s
10,10	s	s	s	s	s	s	s	s	s	s

h: hit, s: stand, p: split, d: double
two letter combination: either action is valid

Table 3: Player Perfect Strategy

Appendix D Player Winning Odds

The following outlines players' winning odds, assuming they follow the strategy outlined in appendix C.

Dealer upcard	Player Odds
2	9.068 %
3	12.278 %
4	15.786 %
5	19.549 %
6	22.752 %
7	14.333 %
8	5.726 %
9	-4.132 %
10	-17.32 %
A	-37.241 %
Total	**-0.859 %**

Table 4: Player odds at playing blackjack on the platform [10]

Appendix E Blackjack Hands per Hour

Players	Hands per hour
1	209
2	139
3	105
4	84
5	70
6	60
7	52
Average	102

Table 5: Blackjack hands played per hour in casinos[14]

Appendix F Acceptance Criteria of Milestones

M0 ERC20 token, crowdsale and DAO contracts

- Implements all methods of the ERC20 interface
- Converts incoming ETH to tokens based on the price of current round
- Refunds ETH when softcap limit is not reached
- Transitions from round 1 to 2 and so on as cap of each round is reached
- Closes the sale when hardcap reaches
- Enforces vesting period of shares, 12 months, unless purchased as part of the sale
- Supports closure of sale by administrator
- Supports assignment of bonus tokens by administrator

- Enables administrators to create and close voting requests
- Enables token holders to cast votes

M1 Investor website with KYC registration

- Provides information about the token sale including link to latest version of whitepaper
- Allows investors to register and receive an investor registration #
- Provides guidance on rules and regulations for investor's country of residence
- Displays the current round, and amount of tokens raised

M2 UX Study

- 5 player and 5 dealer participants aged 21-40
- Study must run the following scenarios using sketches and mockups and provide footage for post-mortem inspection and analysis
  - Player being the only player on the table
  - Player being at a full table
  - Dealer running a game with a single player
  - Dealer running a game with several players
  - Dealer running a game with a single player who occupies several spots on the table

M3 Prototype Demo

- Shows experience of starting a new table as dealer
- Shows experience of searching for a table as a player
- Shows experience of joining a table
- Shows gameplay from both player and dealer perspective

M4 STO Launch

- ERC20 token, crowdsale, and DAO contracts are deployed to the main Ethereum network
- Investor website provides address of the crowdsale contract
- Investor website allows purchase of funds via MetaMask

M5 Beta Launch on Ropsten Network

- Dealers and players can go to cashier and exchange Ether with casino chips
- Dealers can start tables with preset minimum and maximum bet amounts
- System enforces solvency of tables
- Players can search and join tables
- Players can chat with each other in real-time when at a table
- Players can observe tables without needing an account

M6 Customer Feedback

M7 Code Complete

M8 Launch on main Ethereum Network

- Cashier supports integration with ShapeShift

- Dealers can customize minimum and maximum bet amounts when creating tables
- Players can challenge hands
- Players and dealers can review each other
- Dealer profile page with user rankings

M9 Listing on Exchange

M10 Poker Cash Games

- Dealer button rotates between players after every hand
- Small and big blind bets are automatically posted on behalf of players
- Dealer receives instruction to draw pre-flop, flop, turn and river cards
- New players joining can decide to post blind and join immediately, or wait for the big blind
- When all other players fold, the winning player can choose to instruct the dealer to show one, both, or none of his/her cards
- Players can start a straddle prior to beginning of a hand
- Players can search for games based on small and big blind bet amonuts
- Players can tip the dealer at the end of each hand
- Players can request a shuffle of decks

M11 Android and iOS Launch

- Players are able to search for, and join tables
- Players can exchange Ether for chips and vice versa through integration with Coinbase
- Players can submit ratings on dealers
- Players can mute the video feed during gameplay
- Players can hide the chat during gameplay
- Players can observe tables without participating

M12 Poker Tournaments

- Admins can initiate new tournaments
- Players can see a list of upcoming tournaments and sign up by paying the buy-in
- Dealers can see a list of upcoming tournaments and sign up to participate
- Rakes and blinds go up every 5 minutes
- Each tournament displays a list of all players with chip balance, how many are still in the game, and how many have been knocked out
- Players are automatically swapped between tables when a table has 5 players or less
- Tournament payout, rake, rake and blind increase rate is displayed to players prior to tournament start
- System releases dealers as tournament progresses and fewer tables are left. Released dealers may run tables for other games or sign off the platform
- Players and dealers can request an auditor if they observe a problem with how a hand is played. Auditors are notified when this occurs and can inspect the video feed and hand history to make a decision

- Tournament goes on scheduled breaks based on how many players and dealers are left and how long they've been active without a break

M13 Baccarat

- Players can choose the type of games they want to search for, Baccarat must be supported in addition to Blackjack
- Betting can bet on banker, player, player obtaining a pair, or banker obtaining a pair
- Dealer can pull 4 cards, 2 for banker and 2 for player, with additional one card if total is less than 5
- Displays bead plate, small eye, big road, small road, cockroach pig, and big eye boy score cards
- Banker bet wins payout 95% of the wager
- Pair bets pay 8-to-1

Appendix G Poker Pot Sizes

Robinhood conducted a study of hands played on PokerStars.com platform. 16 poker cash game tables were observed on October 12th, for a period of 1 hour. The blinds were \$1/\$2 for all tables observed. The data follows.

Players	Average Pot	Players/Flop %	Hands/hour
7	\$21	16	98
8	\$22	9	100
9	\$45	21	64
5	\$21	12	106
4	\$30	16	120
5	\$20	10	109
5	\$46	9	83
6	\$25	22	70
6	\$20	22	117
6	\$20	12	108
6	\$22	17	104
6	\$17	23	76
6	\$25	20	95
6	\$36	18	115
6	\$20	20	82
6	\$38	20	108
Averages	**\$27**	**16.6875**	**97.1875**

Table 6: Poker Table Observations

Appendix H Hours Spent Per Week by Casino Gamers

The following table highlights results of a survey conducted by Statista that inquires about hours of gameplay spent per week by casino gamers in USA. Using these numbers, we weighted average of spent by gamer is 4.065 hours per week.

Survey Value	Respondents	Proxy Value (in hours)
<1 hours	21%	0.5
1-2 hours	24%	1.5
2-5 hours	25%	3.5
5-10 hours	13%	7.5
10-15 hours	8%	12.5
>15 hours	5%	15
Don't Know	4%	0
Weighted Average		**4.065**

Table 7: Survey Results - Hours Spent by Casino Gamers[21]

Appendix I Token Source Code

I.1 Contract

```
pragma solidity ^0.5.0;

import './EIP20.sol';

/// @author robinhood.casino
/// @title Robinhood (RHC) ERC20 token
contract RHC is EIP20 {

  /// @notice reports number of tokens that are promised to vest in a future date
  uint256 public pendingGrants;

  /// @notice raised when tokens are issued for an account
  event Issuance(address indexed _beneficiary, uint256 _amount);

  struct Grant {
    /// number of shares in the grant
    uint256 amount;
    /// a linux timestamp of when shares can be claimed
    uint vestTime;
    /// whether the claim has been cancelled by admins
    bool isCancelled;
    /// whether the grant has been claimed by the user
    bool isClaimed;
  }

  /// @dev token balance of all addresses
  mapping (address => uint256) private _balances;

  /// @dev tracks who can spend how much.
  mapping (address => mapping (address => uint256)) private _allowances;
```

```
/// @dev balance of tokens that are not vested yet
mapping (address => Grant[]) private _grants;

// used for access management
address private _owner;
mapping (address => bool) private _admins;

constructor(address admin) public {
  _owner = admin;
}

/// @notice name of the Robinhood token
function name() public pure returns (string memory) {
  return "Robinhood";
}

/// @notice symbol of the Robinhood token
function symbol() public pure returns (string memory) {
  return "RHC";
}

/// @notice RHC does not allow breaking up of tokens into fractions.
function decimals() public pure returns (uint8) {
  return 0;
}

modifier onlyAdmins() {
  require(msg.sender == _owner || _admins[msg.sender] == true, "only admins can
  ↪  invoke this function");
  _;
}

/// @dev registers a new admin
function addAdmin(address admin) public onlyAdmins() {
  _admins[admin] = true;
}

/// @dev removes an existing admin
function removeAdmin(address admin) public onlyAdmins() {
  require(admin != _owner, "owner can't be removed");
  delete _admins[admin];
}

/// @dev Gets the balance of the specified address.
/// @param owner The address to query the balance of.
/// @return A uint256 representing the amount owned by the passed address.
function balanceOf(address owner) public view returns (uint256) {
    return _balances[owner];
}
```

```
/// @dev Function to check the amount of tokens that an owner allowed to a
↪  spender.
/// @param owner address The address which owns the funds.
/// @param spender address The address which will spend the funds.
/// @return A uint256 specifying the amount of tokens still available for the
↪  spender.
function allowance(address owner, address spender) public view returns (uint256)
↪  {
    return _allowances[owner][spender];
}

/// @dev Transfer token to a specified address.
/// @param to The address to transfer to.
/// @param value The amount to be transferred.
function transfer(address to, uint256 value) public returns (bool success) {
  require(to != address(0), "Can't transfer tokens to address 0");
  require(balanceOf(msg.sender) >= value, "You don't have sufficient balance to
  ↪  move tokens");

  _move(msg.sender, to, value);

  return true;
}

/// @dev Approve the passed address to spend the specified amount of tokens on
↪  behalf of msg.sender.
/// Beware that changing an allowance with this method brings the risk that
↪  someone may use both the old
/// and the new allowance by unfortunate transaction ordering. One possible
↪  solution to mitigate this
/// race condition is to first reduce the spender's allowance to 0 and set the
↪  desired value afterwards:
/// https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
/// @param spender The address which will spend the funds.
/// @param value The amount of tokens to be spent.
function approve(address spender, uint256 value) public returns (bool success) {
  require(spender != address(0), "Can't set allowance for address 0");
  require(spender != msg.sender, "Use transfer to move your own funds");

  _allowances[msg.sender][spender] = value;
  emit Approval(msg.sender, spender, value);
  return true;
}

/// @dev Transfer tokens from one address to another.
/// @param from address The address which you want to send tokens from
/// @param to address The address which you want to transfer to
/// @param value uint256 the amount of tokens to be transferred
function transferFrom(address from, address to, uint256 value) public returns
↪  (bool) {
  require(to != address(0), "Can't transfer funds to address 0");
```

```
  // Validate that the sender is allowed to move funds on behalf of the owner
  require(allowance(from, msg.sender) >= value, "You're not authorized to
  ↪  transfer funds from this account");
  require(balanceOf(from) >= value, "Owner of funds does not have sufficient
  ↪  balance");

  // Decrease allowance
  _allowances[from][msg.sender] -= value;

  // Move actual token balances
  _move(from, to, value);

  return true;
}

/// @notice cancels all grants pending for a given beneficiary. If you want to
↪  cancel a single
/// vest, cancel all pending grants, and reinstate the ones you plan to keep
function cancelGrants(address beneficiary) public onlyAdmins() {
  Grant[] storage userGrants = _grants[beneficiary];
  for (uint i = 0; i < userGrants.length; i++) {
    Grant storage grant = userGrants[i];
    if (!grant.isCancelled && !grant.isClaimed) {
      grant.isCancelled = true;

      // remove from pending grants
      pendingGrants -= grant.amount;
    }
  }
}

/// @notice Converts a vest schedule into actual shares. Must be called by the
↪  beneficiary
// to convert their vests into actual shares
function claimGrant() public {
  Grant[] storage userGrants = _grants[msg.sender];
  for (uint i = 0; i < userGrants.length; i++) {
    Grant storage grant = userGrants[i];
    if (!grant.isCancelled && !grant.isClaimed && now >= grant.vestTime) {
      grant.isClaimed = true;

      // remove from pending grants
      pendingGrants -= grant.amount;

      // issue tokens to the user
      _issue(msg.sender, grant.amount);
    }
  }
}
```

```
/// @notice returns information about a grant that user has. Returns a tuple
↪  indicating
/// the amount of the grant, when it will vest, whether it's been cancelled, and
↪  whether it's been claimed
/// already.
/// @param grantIndex a 0-based index of user's grant to retrieve
function getGrant(address beneficiary, uint grantIndex) public view returns
↪  (uint, uint, bool, bool) {
  Grant[] storage grants = _grants[beneficiary];
  if (grantIndex < grants.length) {
    Grant storage grant = grants[grantIndex];
    return (grant.amount, grant.vestTime, grant.isCancelled, grant.isClaimed);
  } else {
    revert("grantIndex must be smaller than length of grants");
  }
}

/// @notice returns number of grants a user has
function getGrantCount(address beneficiary) public view returns (uint) {
  return _grants[beneficiary].length;
}

/// @dev Internal function that increases the token supply by issuing new ones
/// and assigning them to an owner.
/// @param account The account that will receive the created tokens.
/// @param amount The amount that will be created.
function issue(address account, uint256 amount) public onlyAdmins() {
  require(account != address(0), "can't mint to address 0");
  require(amount > 0, "must issue a positive amount of tokens");
  _issue(account, amount);
}

/// @dev Internal function that grants shares to a beneficiary in a future date.
/// @param vestTime milliseconds since epoch at which time shares can be claimed
function grant(address account, uint256 amount, uint vestTime) public
↪  onlyAdmins() {
  require(account != address(0), "grant to the zero address is not allowed");
  require(vestTime > now, "vest schedule must be in the future");

  pendingGrants += amount;
  _grants[account].push(Grant(amount, vestTime, false, false));
}

/// @dev Internal helper to move balances around between two accounts.
function _move(address from, address to, uint256 value) private {
  _balances[from] -= value;
  _balances[to] += value;
  emit Transfer(from, to, value);
}

/// @dev issues/mints new tokens for the specified account
```

```
  function _issue(address account, uint256 amount) private {
    totalSupply += amount;
    _balances[account] += amount;
    emit Issuance(account, amount);
  }
}
```

I.2 Tests

```
const RHC = artifacts.require("./RHC.sol");
const helper = require("./utils");

contract("RHC", accounts => {
  var token;
  var owner;

  /**
   * Verifies that the ERC20 Transfer event has been emitted with provided values
   * @param {*} transaction reference to the transaction on the blockchain
   * @param {*} from address from which transfer was initiated
   * @param {*} to address to which tokens were sent
   * @param {*} value amount of tokens transferred
   */
  function verifyTransferEvent(transaction, from, to, value) {
    assert.equal(transaction.logs[0].event, "Transfer");
    const eventArguments = transaction.logs[0].args;
    assert.equal(eventArguments._value.toNumber(), value);
    assert.equal(eventArguments._from, from);
    assert.equal(eventArguments._to, to);
  }

  /**
   * Private helper that successfully mints tokens from the owner account
   * @param {*} address where to send the tokens to
   * @param {*} amount amount of tokens to mint
   */
  async function issue(address, amount) {
    await token.issue(address, amount, { from: accounts[9] });
  }

  beforeEach(async () => {
    owner = accounts[9];
    token = await RHC.new(owner);
  });

  it("should return the name and symbol correctly", async () => {
    assert.equal(
      await token.name().valueOf(),
      "Robinhood",
      "Name is not reported correctly."
    );
```

```
  assert.equal(
    await token.symbol().valueOf(),
    "RHC",
    "Symbol is not correctly reported."
  );
  assert.equal(
    await token.decimals().valueOf(),
    0,
    "RHC does not allow breaking up of tokens."
  );
});

describe("minting", () => {
  it("minting increases an account's balance and total token supply", async () =>
  ↪  {
    assert.equal(
      await token.totalSupply(),
      0,
      "Supply must start with 0 tokens"
    );
    assert.equal(
      await token.balanceOf(accounts[0]),
      0,
      "Account has no tokens minted"
    );

    // Mint some tokens for a particular account
    await token.issue(accounts[0], 100, { from: accounts[9] });

    assert.equal(
      await token.totalSupply(),
      100,
      "Supply must correctly report minted tokens"
    );
    assert.equal(
      await token.balanceOf(accounts[0]),
      100,
      "Balance must now have 100 tokens"
    );
    assert.equal(
      await token.balanceOf(accounts[1]),
      0,
      "Balance of other accounts must be untouched"
    );
  });

  it("only the owner can mint tokens", async () => {
    const ownedby5 = await RHC.new(accounts[5]);

    await helper.expectError(
      ownedby5.issue(accounts[0], 1, { from: accounts[1] }),
```

```
      "We expect this to fail because only the owner can mint"
    );

    // This should succeed because of the explicitly provided super admin
    await ownedby5.issue(accounts[0], 1, { from: accounts[5] });
  });

  it("other admins can mint as well as the owner", async () => {
    const ownedby5 = await RHC.new(accounts[5]);

    // account 5 itself should be able to mint
    await ownedby5.issue(accounts[9], 1, { from: accounts[5] });

    await helper.expectError(
      ownedby5.issue(accounts[1], 1, { from: accounts[1] }),
      "account 1 should not be able to mint because it's not an admin"
    );

    // make account 1 an admin now and then check that they can mint
    await ownedby5.addAdmin(accounts[1], { from: accounts[5] });
    await ownedby5.issue(accounts[1], 1, { from: accounts[1] });
  });
});

describe("transfer by owner", () => {
  it("token holder can't spend more than what they own", async () => {
    await issue(accounts[0], 100);
    // Balance should be 90 after this
    assert.isTrue(
      (await token.transfer(accounts[1], 10, { from: accounts[0] })).receipt
        .status
    );
    await helper.expectError(
      token.transfer(accounts[2], 100, { from: accounts[0] }),
      "should throw because account has insufficient balance"
    );
  });

  it("transfer switches balance between accounts", async () => {
    await issue(accounts[0], 100);
    await token.transfer(accounts[1], 10, { from: accounts[0] });
    await token.transfer(accounts[2], 20, { from: accounts[0] });
    await token.transfer(accounts[3], 30, { from: accounts[0] });

    assert.equal(await token.balanceOf(accounts[0]), 40);
    assert.equal(await token.balanceOf(accounts[1]), 10);
    assert.equal(await token.balanceOf(accounts[2]), 20);
    assert.equal(await token.balanceOf(accounts[3]), 30);

    // transfer back from accounts 3 to 0
    await token.transfer(accounts[0], 30, { from: accounts[3] });
```

```
    assert.equal(await token.balanceOf(accounts[0]), 70);
    assert.equal(await token.balanceOf(accounts[3]), 0);
  });

  it("transfer emits events", async () => {
    await issue(accounts[0], 20);

    const successfulTransfer = await token.transfer(accounts[1], 10, {
      from: accounts[0]
    });
    verifyTransferEvent(successfulTransfer, accounts[0], accounts[1], 10);
  });

  it("should not increase token supply when transfer is executed", async () => {
    await issue(accounts[0], 100);
    assert.equal(await token.totalSupply(), 100);
    await token.transfer(accounts[1], 50, { from: accounts[0] });
    assert.equal(await token.balanceOf(accounts[0]), 50);
    assert.equal(await token.balanceOf(accounts[1]), 50);
    assert.equal(await token.totalSupply(), 100);
  });
});

describe("access control", () => {
  it("only admins can add new admins", async () => {
    const ownedby4 = await RHC.new(accounts[4]);
    await helper.expectError(
      ownedby4.addAdmin(accounts[1], { from: accounts[1] }),
      "Only account 4 must be able to add an admin"
    );
  });

  it("only admins can remove admins", async () => {
    const customAdmin = await RHC.new(accounts[3]);
    await customAdmin.addAdmin(accounts[4], { from: accounts[3] });
    await helper.expectError(
      customAdmin.removeAdmin(accounts[4], { from: accounts[6] }),
      "only accounts 3 and 4 must be able to remove admins"
    );
  });

  it("admins can be removed", async () => {
    const customAdmin = await RHC.new(accounts[3]);
    await customAdmin.addAdmin(accounts[4], { from: accounts[3] });
    await customAdmin.addAdmin(accounts[5], { from: accounts[3] });

    // mint using accounts 3 and 4 since they are now admins
    await customAdmin.issue(accounts[1], 1, { from: accounts[4] });
    await customAdmin.issue(accounts[1], 1, { from: accounts[5] });
```

```
    // remove 4 as an admin by account 4 itself
    await customAdmin.removeAdmin(accounts[4], { from: accounts[4] });

    await helper.expectError(
      customAdmin.issue(accounts[1], 1, { from: accounts[4] }),
      "account 4 should no longer be able to mint tokens because it's not an
      ↪ admin"
    );

    // remove 5 using the owner
    await customAdmin.removeAdmin(accounts[5], { from: accounts[3] });
    await helper.expectError(
      customAdmin.issue(accounts[1], 1, { from: accounts[5] }),
      "account 5 should no longer be able to mint tokens because it's not an
      ↪ admin"
    );
  });

  it("owner can never be removed from admin lists", async () => {
    const owner = accounts[3];
    const customAdmin = await RHC.new(owner);
    await customAdmin.addAdmin(accounts[4], { from: owner });

    // try removing owner
    await helper.expectError(
      customAdmin.removeAdmin(owner, { from: accounts[4] }),
      "Owner can never be removed"
    );
  });
});

describe("vesting", () => {
  it("grants increase grants balance but don't touch token supply", async () => {
    let tokenSupply = (await token.totalSupply()).toNumber();
    let pendingGrants = (await token.pendingGrants()).toNumber();
    await token.grant(
      accounts[0],
      10,
      (await helper.getCurrentTime()) + 10000,
      { from: owner }
    );
    assert.equal(
      (await token.pendingGrants()).toNumber(),
      pendingGrants + 10,
      "pending grants must increase by 10"
    );
    assert.equal(
      (await token.totalSupply()).toNumber(),
      tokenSupply,
      "token supply must remain unchanged"
    );
```

```
});

it("vestings can only be claimed once, even if attempted multiple times", async
↪  () => {
  const vestDelta = 10000;
  const vestTime = (await helper.getCurrentTime()) + vestDelta;
  await token.grant(accounts[0], 10, vestTime, { from: owner });

  // set time to be after vesting time and request grant
  const tokenBalance = (await token.balanceOf(accounts[0])).toNumber();
  const tokenSupplyBeforeVest = (await token.totalSupply()).toNumber();
  await helper.advanceTimeAndBlock(vestDelta + 1);

  // claim multiple times
  await token.claimGrant({ from: accounts[0] });
  await token.claimGrant({ from: accounts[0] });
  await token.claimGrant({ from: accounts[0] });

  assert.equal(
    (await token.balanceOf(accounts[0])).toNumber(),
    tokenBalance + 10,
    "grants must succeed because vest time has come"
  );
  assert.equal(
    (await token.totalSupply()).toNumber(),
    tokenSupplyBeforeVest + 10
  );
});

it("shares cannot be claimed prior to their vesting schedule", async () => {
  const vestDelta = 100000;
  const vestTime = (await helper.getCurrentTime()) + vestDelta;
  const tokenBalance = (await token.balanceOf(accounts[0])).toNumber();
  await token.grant(accounts[0], 10, vestTime, { from: owner });

  // request vest and expect nothing to happen because vest time has not hit
  ↪  yet
  await helper.advanceTimeAndBlock(vestDelta - 60);
  await token.claimGrant({ from: accounts[0] });

  assert.equal(
    (await token.balanceOf(accounts[0])).toNumber(),
    tokenBalance,
    "request grant should not take effect because time has not come yet."
  );
});

it("shares can be claimed after vesting period elapses", async () => {
  const vestDelta = 10000;
  const vestTime = (await helper.getCurrentTime()) + vestDelta;
  await token.grant(accounts[0], 10, vestTime, { from: owner });
```

```
  // set time to be after vesting time and request grant
  const tokenBalance = (await token.balanceOf(accounts[0])).toNumber();
  const tokenSupplyBeforeVest = (await token.totalSupply()).toNumber();
  await helper.advanceTimeAndBlock(vestDelta + 1);
  await token.claimGrant({ from: accounts[0] });
  assert.equal(
    (await token.balanceOf(accounts[0])).toNumber(),
    tokenBalance + 10,
    "grants must succeed because vest time has come"
  );
  assert.equal(
    (await token.totalSupply()).toNumber(),
    tokenSupplyBeforeVest + 10
  );
});

it("vesting must decrease pending grant balance", async () => {
  const vestDelta = 10000;
  const vestTime = (await helper.getCurrentTime()) + vestDelta;
  await token.grant(accounts[0], 10, vestTime, { from: owner });

  const pendingGrants = (await token.pendingGrants()).toNumber();

  // set time to be after vesting time and request grant
  await helper.advanceTimeAndBlock(vestDelta + 1);
  await token.claimGrant({ from: accounts[0] });

  assert.equal(
    (await token.pendingGrants()).toNumber(),
    pendingGrants - 10,
    "grant balance must decrease when vest is successful"
  );
});

it("user can have multiple pending vests", async () => {
  const now = await helper.getCurrentTime();
  for (var i = 1; i < 4; i++) {
    await token.grant(accounts[1], 10, now + i * 10000, { from: owner });
  }

  assert.equal(
    (await token.pendingGrants()).toNumber(),
    30,
    "30 shares must be pending"
  );

  // execute a claim that should not result in any vests since the time has not
  ↪  come yet
  await token.claimGrant({ from: accounts[1] });
```

```
  assert.equal(
    (await token.pendingGrants()).toNumber(),
    30,
    "30 shares must be pending"
  );

  // set time to when 2 grants vest
  await helper.advanceTimeAndBlock(2 * 10000);
  await token.claimGrant({ from: accounts[1] });

  // at this point we expect two grants to vest and one to be pending
  assert.equal((await token.balanceOf(accounts[1])).toNumber(), 20);
  assert.equal((await token.totalSupply()).toNumber(), 20);
  assert.equal((await token.pendingGrants()).toNumber(), 10);

  // another claim must be a noop
  await token.claimGrant({ from: accounts[1] });
  assert.equal((await token.balanceOf(accounts[1])).toNumber(), 20);
  assert.equal((await token.totalSupply()).toNumber(), 20);
  assert.equal((await token.pendingGrants()).toNumber(), 10);
});

it("vests can only be granted by admins", async () => {
  // expecting this to succeed since admin is invoking it
  await token.grant(
    accounts[0],
    1,
    (await helper.getCurrentTime()) + 20000,
    { from: owner }
  );
  await token.addAdmin(accounts[1], { from: owner });
  await token.grant(
    accounts[0],
    1,
    (await helper.getCurrentTime()) + 20000,
    { from: accounts[1] }
  );

  await helper.expectError(
    token.grant(accounts[0], 1, (await helper.getCurrentTime()) + 30000, {
      from: accounts[0]
    }),
    "only admins must be able to grant tokens"
  );
});

it("grants can't be in the past", async () => {
  const now = await helper.getCurrentTime();
  await helper.expectError(
    token.grant(accounts[0], 1, now, { from: owner }),
    "vest must be in the future"
```

```
  );
  await helper.expectError(
    token.grant(accounts[0], 1, now - 1, { from: owner }),
    "vest must be in the future"
  );
});

it("vests can only be cancelled by admins", async () => {
  const now = await helper.getCurrentTime();
  await token.grant(accounts[1], 10, now + 100000, { from: owner });

  await helper.expectError(
    token.cancelGrants(accounts[1], { from: accounts[2] }),
    "cancelling grants can only be done by admins"
  );

  // expecting this one to go through
  await token.cancelGrants(accounts[1], { from: owner });
});

it("a cancelled vest can't be claimed by the beneficiary", async () => {
  const now = await helper.getCurrentTime();
  await token.grant(accounts[1], 10, now + 10000, { from: owner });

  // cancel the grant
  await token.cancelGrants(accounts[1], { from: owner });

  assert.equal(
    (await token.pendingGrants()).toNumber(),
    0,
    "pending balance of grants must be back at 0"
  );

  // try to claim the grant
  await helper.advanceTimeAndBlock(15000);
  await token.claimGrant({ from: accounts[1] });

  assert.equal(
    (await token.balanceOf(accounts[1])).toNumber(),
    0,
    "claim should have resulted in 0 shares"
  );
});

it("canceling one vest cancels all other pending grants", async () => {
  const now = await helper.getCurrentTime();
  for (var i = 1; i < 4; i++) {
    await token.grant(accounts[1], 10, now + i * 10000, { from: owner });
  }

  await token.cancelGrants(accounts[1], { from: owner });
```

```
  assert.equal((await token.pendingGrants()).toNumber(), 0);
});

it("should return a count of 0 grants for a user without any grants", async ()
↪  => {
  assert.equal((await token.getGrantCount(accounts[1])).toNumber(), 0);
});

it("user has access to all their grants", async () => {
  const now = await helper.getCurrentTime();
  for (var i = 1; i < 4; i++) {
    await token.grant(accounts[1], i * 10, now + i * 10000, {
      from: owner
    });
  }

  // one of the grants should be claimed now
  await helper.advanceTimeAndBlock(15000);
  await token.claimGrant({ from: accounts[1] });

  // cancel remaining grants
  await token.cancelGrants(accounts[1], { from: owner });

  // add a new grant again
  await token.grant(accounts[1], 10, now + 50000, { from: owner });

  // user should have a total of 4 grants, one claimed, 2 cancelled and one
  ↪  pending
  assert.equal((await token.getGrantCount(accounts[1])).toNumber(), 4);

  let grant1 = await token.getGrant(accounts[1], 0);
  assert.equal(grant1[0].toNumber(), 10);
  assert.equal(grant1[1].toNumber(), now + 10000);
  assert.equal(grant1[2], false);
  assert.equal(grant1[3], true);

  let grant2 = await token.getGrant(accounts[1], 1);
  assert.equal(grant2[0].toNumber(), 20);
  assert.equal(grant2[1].toNumber(), now + 20000);
  assert.equal(grant2[2], true);
  assert.equal(grant2[3], false);

  let grant3 = await token.getGrant(accounts[1], 2);
  assert.equal(grant3[0].toNumber(), 30);
  assert.equal(grant3[1].toNumber(), now + 30000);
  assert.equal(grant3[2], true);
  assert.equal(grant3[3], false);

  let grant4 = await token.getGrant(accounts[1], 3);
  assert.equal(grant4[0].toNumber(), 10);
```

```
    assert.equal(grant4[1].toNumber(), now + 50000);
    assert.equal(grant4[2], false);
    assert.equal(grant4[3], false);
  });

  // Ignoring this test because geth has a bug where it doesn't handle revert()
  ↪  in public view functions
  // properly and doesn't detect the aborted execution by the contract. There are
  ↪  no functional defects
  // in the contract code.
  xit("retrieval of non-existing grant results in error", async () => {
    await helper.expectError(
      token.getGrant(accounts[1], 1),
      "didn't return error for invalid grant index"
    );
  });
});

describe("transfer through delegation", () => {
  it("allows spending funds that are approved", async () => {
    // mint for accounts A and B
    await issue(accounts[0], 10);
    await issue(accounts[1], 10);

    // allow account C to transfer from A and B
    await token.approve(accounts[2], 5, { from: accounts[0] });
    await token.approve(accounts[2], 5, { from: accounts[1] });

    // allowance should correctly report set permissions
    assert.equal(await token.allowance(accounts[0], accounts[2]), 5);
    assert.equal(await token.allowance(accounts[1], accounts[2]), 5);

    // move funds to C itself
    await token.transferFrom(accounts[0], accounts[2], 1, {
      from: accounts[2]
    });

    // check that C's balance is increased and its allowance is decreased
    assert.equal(await token.balanceOf(accounts[2]), 1);
    assert.equal(await token.allowance(accounts[0], accounts[2]), 4);

    // move funds to D since C is allowed to move funds anywhere
    await token.transferFrom(accounts[0], accounts[3], 1, {
      from: accounts[2]
    });
    await token.transferFrom(accounts[1], accounts[3], 1, {
      from: accounts[2]
    });

    // check account D balance and ensure allowance of C is properly updated
    assert.equal(await token.balanceOf(accounts[3]), 2);
```

```
  assert.equal(await token.balanceOf(accounts[0]), 8);
  assert.equal(await token.balanceOf(accounts[1]), 9);
  assert.equal(await token.allowance(accounts[0], accounts[2]), 3);
  assert.equal(await token.allowance(accounts[1], accounts[2]), 4);

  // account C's balance must remain unchanged after it transferred to D
  assert.equal(await token.balanceOf(accounts[2]), 1);
});

it("only approved users can transfer tokens", async () => {
  await issue(accounts[0], 10);

  // allow account 5 to move 2 tokens
  await token.approve(accounts[5], 2, { from: accounts[0] });

  // should be OK because 5 is allowed to transfer
  await token.transferFrom(accounts[0], accounts[1], 1, {
    from: accounts[5]
  });

  // should be disallowed because account 4 is not allowed to transfer
  await helper.expectError(
    token.transferFrom(accounts[0], accounts[2], 1, { from: accounts[4] }),
    "disallowed user allowed to perform transfer"
  );

  // user itself should not be allowed to use transferFrom (and should instead
  ↪  use transfer)
  await helper.expectError(
    token.transferFrom(accounts[0], accounts[3], 1, { from: accounts[0] }),
    "same user should use transfer function"
  );
});

it("owner should use direct transfer to move his/her own funds", async () => {
  await issue(accounts[0], 10);
  await helper.expectError(
    token.approve(accounts[0], 1, { from: accounts[0] }),
    "Owner must use the transfer function to move funds"
  );
});

it("transfer works only when user has sufficient balance", async () => {
  await issue(accounts[1], 5);
  await token.approve(accounts[2], 5, { from: accounts[1] });

  // Spend the balance of account 1
  await token.transfer(accounts[3], 4, { from: accounts[1] });

  // Even though user is approved to spend, the call should fail because
  // user does not have sufficient balance
```

```
  await helper.expectError(
    token.transferFrom(accounts[1], accounts[5], 2, { from: accounts[2] }),
    "Delegate can spend tokens even though the owner has no balance."
  );

  // When there is sufficient balance though, the transfer must go through
  ↪  successfully
  await token.transferFrom(accounts[1], accounts[5], 1, {
    from: accounts[2]
  });
  assert.equal(await token.balanceOf(accounts[5]), 1);
});

it("approval emits events", async () => {
  await issue(accounts[1], 5);
  const approval = await token.approve(accounts[2], 5, {
    from: accounts[1]
  });
  assert.equal(approval.logs[0].event, "Approval");
  const eventArguments = approval.logs[0].args;
  assert.equal(eventArguments._owner, accounts[1]);
  assert.equal(eventArguments._spender, accounts[2]);
  assert.equal(eventArguments._value.toNumber(), 5);
});

it("delegated transfer emits events", async () => {
  await issue(accounts[1], 5);
  await token.approve(accounts[2], 5, { from: accounts[1] });
  const transfer = await token.transferFrom(accounts[1], accounts[3], 1, {
    from: accounts[2]
  });
  verifyTransferEvent(transfer, accounts[1], accounts[3], 1);
});

it("allowance is reported when transfer is approved", async () => {
  await issue(accounts[1], 5);
  assert.equal(
    await token.allowance(accounts[1], accounts[2]),
    0,
    "nothing has been approved yet"
  );
  await token.approve(accounts[2], 3, { from: accounts[1] });
  assert.equal(await token.allowance(accounts[1], accounts[2]), 3);
});

it("should decrease allowance as tokens are spent", async () => {
  await issue(accounts[1], 5);
  await token.approve(accounts[2], 3, { from: accounts[1] });
  assert.equal(await token.allowance(accounts[1], accounts[2]), 3);
  await token.transferFrom(accounts[1], accounts[3], 2, {
    from: accounts[2]
```

```
  });
  assert.equal(
    await token.allowance(accounts[1], accounts[2]),
    1,
    "allowance must decrease as tokens are spent"
  );
});

it("should allow multiple withdrawals by same delegate", async () => {
  await issue(accounts[1], 5);
  await token.approve(accounts[2], 5, { from: accounts[1] });
  await token.transferFrom(accounts[1], accounts[3], 1, {
    from: accounts[2]
  });
  await token.transferFrom(accounts[1], accounts[3], 1, {
    from: accounts[2]
  });
  await token.transferFrom(accounts[1], accounts[3], 1, {
    from: accounts[2]
  });
  assert.equal(await token.balanceOf(accounts[3]), 3);
  assert.equal(await token.balanceOf(accounts[1]), 2);
});

it("should allow multiple delegates for the same owner account", async () => {
  await issue(accounts[1], 5);
  await token.approve(accounts[2], 5, { from: accounts[1] });
  await token.approve(accounts[3], 5, { from: accounts[1] });

  // Send tokens from 2 delegates to the same account
  await token.transferFrom(accounts[1], accounts[4], 1, {
    from: accounts[2]
  });
  await token.transferFrom(accounts[1], accounts[4], 1, {
    from: accounts[3]
  });

  assert.equal(await token.balanceOf(accounts[4]), 2);
  assert.equal(await token.balanceOf(accounts[1]), 3);

  // Nobody should be able to overspend, even though they were originally
  ↪  allowed to spend all
  // tokens of account 1
  await helper.expectError(
    token.transferFrom(accounts[1], accounts[4], 4, { from: accounts[2] }),
    "account 4 can't spend because account 1 has insufficient balance"
  );
  await helper.expectError(
    token.transferFrom(accounts[1], accounts[4], 4, { from: accounts[3] }),
    "account 3 can't spend because account 1 has insufficient balance"
  );
```

```
    });

    it("should decrease user's balance when delegated transfer is executed", async
    ↪  () => {
      await issue(accounts[1], 5);
      await token.approve(accounts[2], 3, { from: accounts[1] });
      await token.transferFrom(accounts[1], accounts[2], 3, {
        from: accounts[2]
      });
      assert.equal(await token.balanceOf(accounts[1]), 2);
      assert.equal(await token.balanceOf(accounts[2]), 3);
      assert.equal(await token.allowance(accounts[1], accounts[2]), 0);
    });

    it("should not change token supply when delegated transfer is executed", async
    ↪  () => {
      await issue(accounts[1], 5);
      assert.equal(await token.totalSupply(), 5);
      await token.approve(accounts[2], 3, { from: accounts[1] });
      assert.equal(await token.totalSupply(), 5);
      await token.transferFrom(accounts[1], accounts[2], 3, {
        from: accounts[2]
      });
      assert.equal(await token.totalSupply(), 5);
    });
  });
});
```

Appendix J Crowdsale Source Code

J.1 Contract

```
pragma solidity ^0.5.0;

import "./RHC.sol";

contract Crowdsale {

  /// @dev represents a round of token sale
  struct Round {
    /// @dev price per token for every token
    uint tokenPrice;
    /// @dev total number of tokens available in the round
    uint capacityLeft;
  }

  /// @notice event is raised when a token sale occurs
  /// @param amountSent amount of money sent by the purchaser
  /// @param amountReturned amount of money returned to the purchaser in case
  ↪  amount sent was not exact
  /// @param buyer the address which purchased the tokens
```

```
event Sale(uint amountSent, uint amountReturned, uint tokensSold, address buyer);

/// @notice raised when all tokens are sold out
event SaleCompleted();

/// @notice raised when a round completes and the next round starts
/// @param oldTokenPrice previous price per token
/// @param newTokenPrice new price per token
event RoundChanged(uint oldTokenPrice, uint newTokenPrice);

/// @dev information about rounds of fundraising in the crowdsale
Round[] private _rounds;
uint8 private _currentRound;

/// @notice where the contract wires funds in exchange for tokens
address payable private wallet;

/// @notice a refenence to the RHC token being sold
RHC public token;

/// @notice reports whether the sale is still open
bool public isSaleOpen;

/// @dev how much wei has been raised so far
uint public weiRaised;

/// @dev how many tokens have been sold so far
uint public tokensSold;

/// @notice creates the crowdsale. Only intended to be used by Robinhood team.
constructor(address payable targetWallet, uint[] memory roundPrices, uint[]
↪  memory roundCapacities) public {
  require(roundPrices.length == roundCapacities.length, "Equal number of round
  ↪  parameters must be specified");
  require(roundPrices.length >= 1, "Crowdsale must have at least one round");
  require(roundPrices.length < 10, "Rounds are limited to 10 at most");

  // store rounds
  _currentRound = 0;
  for (uint i = 0; i < roundPrices.length; i++) {
    _rounds.push(Round(roundPrices[i], roundCapacities[i]));
  }

  wallet = targetWallet;
  isSaleOpen = true;
  weiRaised = 0;
  tokensSold = 0;

  // Create token with this contract as the owner
  token = new RHC(address(this));
```

```
  // add target wallet as an additional owner
  token.addAdmin(wallet);
}

function() external payable {
  uint amount = msg.value;
  address payable buyer = msg.sender;
  require(amount > 0, "must send money to get tokens");
  require(buyer != address(0), "can't send from address 0");
  require(isSaleOpen, "sale must be open in order to purchase tokens");

  (uint tokenCount, uint change) = calculateTokenCount(amount);

  // if insufficient money is sent, return the buyer's mone
  if (tokenCount == 0) {
    buyer.transfer(change);
    return;
  }

  // this is how much of the money will be consumed by this token purchase
  uint acceptedFunds = amount - change;

  // forward funds to owner
  wallet.transfer(acceptedFunds);

  // return left over (unused) funds back to the sender
  buyer.transfer(change);

  // assign tokens to whoever is purchasing
  token.issue(buyer, tokenCount);

  // update state tracking how much wei has been raised so far
  weiRaised += acceptedFunds;
  tokensSold += tokenCount;

  updateRounds(tokenCount);

  emit Sale(amount, change, tokenCount, buyer);
}

/// @notice given an amount of money returns how many tokens the money will
↪  result in with the
/// current round's pricing
function calculateTokenCount(uint money) public view returns (uint count, uint
↪  change) {
  require(isSaleOpen, "sale is no longer open and tokens can't be purchased");

  // get current token price
  uint price = _rounds[_currentRound].tokenPrice;
  uint capacityLeft = _rounds[_currentRound].capacityLeft;
```

```
    // money sent must be bigger than or equal the price, otherwise, no purchase is
    ↪  necessary
    if (money < price) {
      // return all the money
      return (0, money);
    }

    count = money / price;
    change = money % price;

    // Ensure there's sufficient capacity in the current round. If the user wishes
    ↪  to
    // purchase more, they can send money again to purchase tokens at the next
    ↪  round
    if (count > capacityLeft) {
      change += price * (count - capacityLeft);
      count = capacityLeft;
    }

    return (count, change);
  }

  /// increases the round or closes the sale if tokens are sold out
  function updateRounds(uint tokens) private {
    Round storage currentRound = _rounds[_currentRound];
    currentRound.capacityLeft -= tokens;

    if (currentRound.capacityLeft <= 0) {
      if (_currentRound == _rounds.length - 1) {
        isSaleOpen = false;
        emit SaleCompleted();
      } else {
        _currentRound++;
        emit RoundChanged(currentRound.tokenPrice,
        ↪  _rounds[_currentRound].tokenPrice);
      }
    }
  }
}
```

J.2 Tests

```
const Crowdsale = artifacts.require("./Crowdsale.sol");
const RHC = artifacts.require("./RHC.sol");
const helper = require("./utils");

contract("Crowdsale", accounts => {
  var sale;
  var token;
  let owner = accounts[9];
  let advisors = accounts[8];
```

```
let legal = accounts[7];
let developers = accounts[6];
let founders = accounts[5];
let reserve = accounts[4];

const _1_e18 = "1000000000000000000";
const _2_e18 = "2000000000000000000";
const _3_e18 = "3000000000000000000";
const gasPriceInWei = 20000000000;

beforeEach(async () => {
  sale = await Crowdsale.new(
    owner,
    // passing in values as string to workaround
    // https://github.com/ethereum/web3.js/issues/2077
    [_1_e18, _2_e18, _3_e18],
    [3, 4, 5]
  );
  token = await RHC.at(await sale.token());
});

async function deployContract(roundPrices, roundCapacities) {
  return await Crowdsale.new(
    owner,
    roundPrices,
    roundCapacities
  );
}

async function grantsVestAtOrAfterTime(address, time) {
  let grantCount = (await token.getGrantCount(address)).toNumber();
  // If the user has no grants, then return false immediately
  if (grantCount == 0) {
    return false;
  }
  for (let i = 0; i < grantCount; i++) {
    let grant = await token.getGrant(address, i);
    // if the grant vests prior to the specified time, then return false
    if (grant[1].toNumber() < time) {
      return false;
    }
  }
  // all grants vest after the specified time
  return true;
}

describe("balance sanity", () => {
  it("results in 0 grants for initial members", async () => {
    assert.equal((await token.pendingGrants()).toNumber(), 0);
  });
```

```
it("sale must be open at start", async () => {
  assert.isTrue(await sale.isSaleOpen());
});

it("wei raised must be 0 at the start", async () => {
  assert.equal((await sale.weiRaised()).toNumber(), 0);
});

it("token sold must be 0 at the start", async () => {
  assert.equal((await sale.tokensSold()).toNumber(), 0);
});

it("money sent to purchase tokens is forwarded to the crowdsale owner", async
↪  () => {
  let balanceBefore = await helper.getBalance(owner);
  // buy some tokens
  await sale.send(1e18, {
    from: accounts[0]
  });
  let balanceAfter = await helper.getBalance(owner);
  assert.equal(balanceAfter, balanceBefore + 1e18);
});

it("increases wei raised and tokens sold balances when a sale occurs", async ()
↪  => {
  let supplyBefore = (await token.totalSupply()).toNumber();
  let weiRaisedBefore = await sale.weiRaised();
  let tokensSoldBefore = (await sale.tokensSold()).toNumber();
  await sale.send(1e18, {
    from: accounts[0]
  });
  let weiRaisedAfter = await sale.weiRaised();
  let tokensSoldAfter = (await sale.tokensSold()).toNumber();
  let supplyAfter = (await token.totalSupply()).toNumber();
  assert.equal(weiRaisedAfter.sub(weiRaisedBefore).toString(), _1_e18);
  assert.equal(tokensSoldAfter, tokensSoldBefore + 1);
  assert.equal(supplyAfter, supplyBefore + 1);
});

it("does not grants initial members any tokens", async () => {
  const now = await helper.getCurrentTime();
  const inAYear = now + 60 * 60 * 24 * 365;
  assert.isFalse(await grantsVestAtOrAfterTime(founders, inAYear));
  assert.isFalse(await grantsVestAtOrAfterTime(developers, inAYear));
  assert.isFalse(await grantsVestAtOrAfterTime(advisors, inAYear));
  assert.isFalse(await grantsVestAtOrAfterTime(legal, inAYear));
  assert.isFalse(await grantsVestAtOrAfterTime(owner, inAYear));
});

it("token balance of buyer in the token contract is increased when a sale
↪  occurs", async () => {
```

```
    await sale.send(1e18, {
      from: accounts[0]
    });
    assert.equal((await token.balanceOf(accounts[0])).toNumber(), 1);
  });
});

describe("rounds and progress of the sale", () => {
  it("sending the crowdsale money must result in eventual completion of the
  ↪  sale", async () => {
    // buy out all of round 1
    await sale.send(3 * 1e18, {
      from: accounts[0]
    });
    // buy out all of round 2
    await sale.send(4 * 2e18, {
      from: accounts[1]
    });
    // buy out all of round 3
    await sale.send(5 * 3e18, {
      from: accounts[2]
    });
    assert.isFalse(
      await sale.isSaleOpen(),
      "all tokens are sold out by this time and sale must be closed"
    );
    assert.equal(
      (await sale.tokensSold()).toNumber(),
      12,
      "there are a total of 12 tokens to be sold"
    );
    assert.equal(
      (await sale.weiRaised()).toString(),
      "26000000000000000000",
      "all the money should have been accepted"
    );
  });

  it("buying all tokens of the last round closes the sale", async () => {
    let crowdsale = await deployContract([_1_e18], [1]);
    await crowdsale.send(1e18, {
      from: accounts[2]
    });
    assert.isFalse(await crowdsale.isSaleOpen());
  });
});

describe("event sanity", () => {
  it("emits sale event when tokens are successfully purchased", async () => {
    // buy some tokens
    let transaction = await sale.send(1e18, {
```

```
      from: accounts[0]
    });
    assert.equal(transaction.logs[0].event, "Sale");
    const eventArguments = transaction.logs[0].args;
    // event Sale(uint amountSent, uint amountReturned, uint tokensSold, address
    ↪  buyer);
    assert.equal(eventArguments.amountSent.toString(), _1_e18);
    assert.equal(eventArguments.amountReturned.toNumber(), 0);
    assert.equal(eventArguments.tokensSold.toNumber(), 1);
    assert.equal(eventArguments.buyer, accounts[0]);
  });

  it("emits events when round completes", async () => {
    await sale.send(1e18, {
      from: accounts[1]
    });
    await sale.send(1e18, {
      from: accounts[2]
    });
    let transaction = await sale.send(1e18, {
      from: accounts[3]
    });

    // transaction 3 should result in a round changed event since all tokens are
    ↪  bought
    assert.equal(transaction.logs[0].event, "RoundChanged");
    const eventArguments = transaction.logs[0].args;
    // event RoundChanged(uint oldTokenPrice, uint newTokenPrice);
    assert.equal(eventArguments.oldTokenPrice.toString(), _1_e18);
    assert.equal(eventArguments.newTokenPrice.toString(), _2_e18);
  });

  it("emits sale completed event", async () => {
    let crowdsale = await deployContract([_1_e18], [1]);
    let transaction = await crowdsale.send(1e18, {
      from: accounts[0]
    });
    assert.equal(transaction.logs[0].event, "SaleCompleted");
  });
});

describe("returns extra money", () => {
  it("extra money is returned when the total balance exceeds the leftovers of a
  ↪  round", async () => {
    // send 5 ETH, but only 3 ETH can be used to finish round 1, so 2 ETH must be
    ↪  sent back
    const balanceBefore = await helper.getBalanceInBN(accounts[1]);
    const purchase = await sale.send(5e18, {
      from: accounts[1]
    });
    const balanceAfter = await helper.getBalanceInBN(accounts[1]);
```

```
  // should get 2 ETH back
  assert.isTrue(
    balanceAfter.eq(
      balanceBefore
      .sub(new web3.utils.BN(_3_e18, 10))
      .sub(
        new web3.utils.BN(
          (purchase.receipt.gasUsed * gasPriceInWei).toString(),
          10
        )
      )
    )
  );
});

it("sending money less than the token price should return the money back to the
↪  buyer", async () => {
  let weiRaisedBefore = await sale.weiRaised();
  let tokensSoldBefore = await sale.tokensSold();
  let balanceBefore = await helper.getBalanceInBN(accounts[0]);
  let purchaseAttempt = await sale.send(1e17, {
    from: accounts[0]
  });
  let balanceAfter = await helper.getBalanceInBN(accounts[0]);
  let weiRaisedAfter = await sale.weiRaised();
  let tokensSoldAfter = await sale.tokensSold();
  assert.equal(weiRaisedAfter.toString(), weiRaisedBefore.toString());
  assert.equal(tokensSoldBefore.toString(), tokensSoldAfter.toString());
  // Just some money wasted on gas
  assert.isTrue(
    balanceAfter.eq(
      balanceBefore.sub(
        new web3.utils.BN(
          (purchaseAttempt.receipt.gasUsed * gasPriceInWei).toString(),
          10
        )
      )
    )
  );
});

it("returns unused funds to the buyer", async () => {
  let crowdsale = await deployContract([_1_e18], [3]);
  let balanceBefore = await helper.getBalanceInBN(accounts[0]);
  let purchase = await crowdsale.send(15e17, {
    from: accounts[0]
  });
  // transaction should return 0.5 ETH back
  let balanceAfter = await helper.getBalanceInBN(accounts[0]);
  // balance should be 0.5 ETH minus how much the transaction cost (gas used *
  ↪  gas price)
```

```
    assert.isTrue(
      balanceAfter.eq(
        balanceBefore
        .sub(new web3.utils.BN(_1_e18, 10))
        .sub(
          new web3.utils.BN(
            (purchase.receipt.gasUsed * gasPriceInWei).toString(),
            10
          )
        )
      )
    );
  });
});

describe("gas usage", () => {
  it("costs reasonable amounts of gas to call", async () => {
    // the following transaction returns some funds back to the buyer, so it's a
    ↪  relatively expensive
    // operation to call. Its gas use should be representative of what users need
    ↪  to pay
    let crowdsale = await deployContract([_1_e18], [3]);
    let purchase = await crowdsale.send(15e17, {
      from: accounts[0]
    });

    console.log(`Gas usage for share purchase: ${purchase.receipt.gasUsed}`);
  })
});

describe("token and change calculation", () => {
  it("reports 0 token count when insufficient money is supplied", async () => {
    let crowdsale = await deployContract([_1_e18], [3]);

    // 0.1 ethers should give me no shares and all the money back
    let __1e17 = new web3.utils.BN("100000000000000000", 10);
    let result = await crowdsale.calculateTokenCount(__1e17);
    assert.equal(
      result[0].toNumber(),
      0,
      "less money than the token price is supplied"
    );
    assert.isTrue(__1e17.eq(result[1]));
  });

  it("returns change when supplied money is not divisible by token price", async
  ↪  () => {
    let crowdsale = await deployContract([_1_e18], [3]);

    let __15e17 = new web3.utils.BN("150000000000000000", 10);
    let __5e17 = new web3.utils.BN("500000000000000000", 10);
```

```
      let result = await crowdsale.calculateTokenCount(__15e17);
      assert.equal(result[0].toNumber(), 1);
      assert.isTrue(__5e17.eq(result[1]));
    });

    it("returns no change when money is an exact match of token price", async () =>
    ↪  {
      let crowdsale = await deployContract([_1_e18], [3]);
      let twice = new web3.utils.BN(_1_e18, 10).mul(new web3.utils.BN("2", 10));

      let result = await crowdsale.calculateTokenCount(twice);
      assert.equal(result[0].toNumber(), 2);
      assert.equal(result[1].toNumber(), 0);
    });
  });
});
```

Appendix K DAO Vote Source Code

K.1 Contract

```
pragma solidity ^0.5.0;

import "./RHC.sol";

/// @author robinhood.casino
/// @title A voting contract that allows shareholders to vote on various business
↪  decisions, for example
/// on how much to raise the fee the card dealers are charged.
/// @notice The voting choices are tracked as integers to keep the gas cost of the
↪  vote low for participants.
/// the contract only knows how many choices there are but doesn't understand the
↪  choices themselves. Robinhood
/// admins will explain the meaning of choices to shareholders when they
↪  instantiate a vote. For example,
/// choice 0 means to raise the fee to 10%, and choice 1 means to raise the fee to
↪  20%.
contract Vote {
  /// reference to the token for determining voting rights
  RHC private token;

  /// @dev number of choices available for voters to choose from
  uint public numChoices;

  /// @notice number of votes cast by shareholders
  uint256 public voteCount;

  /// @dev tracks how many votes each choice has received
  mapping (uint => uint256) private votes;

  /// @dev tracks who has voted so far to prevent double voting
```

```
  mapping (address => bool) private voters;

  /// @notice this event is raised when a shareholder casts a vote. The weight
  ↪  parameter is determined
  /// based on share ownership and indicates with what weight the vote is counted
  event VoteCast(address shareholder, uint choice, uint256 weight);

  constructor(RHC rhc, uint choices) public {
    require(address(rhc) != address(0), "must provide a valid RHC token address");
    require(choices > 0, "must have at least one choice to vote on");
    token = rhc;
    numChoices = choices;
  }

  /// @notice casts a vote for a given choice
  function cast(uint choice) public {
    require(choice < numChoices, "invalid choice to vote on");
    require(voters[msg.sender] == false, "you've already cast your vote, can't vote
    ↪  twice");

    address voter = msg.sender;
    uint256 weight = token.balanceOf(voter);

    require(weight > 0, "you don't own any tokens and therefore can't vote");
    require(weight > 1199, "you need to own at least 1200 shares to vote");

    // track the fact that vote has been cast
    voters[voter] = true;
    voteCount++;
    votes[choice] += weight;

    emit VoteCast(voter, choice, weight);
  }

  function getVoteCount(uint choice) public view returns (uint256) {
    require(choice < numChoices, "must ask for a valid vote choice");
    return votes[choice];
  }
}
```

K.2 Tests

```
const Vote = artifacts.require("./Vote.sol");
const RHC = artifacts.require("./RHC.sol");
const helper = require("./utils");

contract("Vote", accounts => {
  var owner;
  var token;
  var vote;
```

```
beforeEach(async () => {
  owner = accounts[9];
  token = await RHC.new(owner);
  vote = await Vote.new(token.address, 3);
});

it("doesn't let anyone who does not own any shares to vote", async () => {
  await helper.expectError(
    vote.cast(0, { from: accounts[1] }),
    "We expect this to fail because accounts[1] does not own any shares"
  );
});

it("requires a minimum of 1200 shares to vote", async () => {
  await token.issue(accounts[0], 1200, { from: owner });
  await vote.cast(0, { from: accounts[0] });
  assert.equal((await vote.voteCount()).toNumber(), 1);
  assert.equal((await vote.getVoteCount(0)).toNumber(), 1200);

  await token.issue(accounts[1], 1199, { from: owner });
  await helper.expectError(
    vote.cast(0, { from: accounts[1] }),
    "need at least 1200 shares to vote"
  );
});

it("tracks votes proportional to share ownership", async () => {
  await token.issue(accounts[0], 1200, { from: owner });
  await token.issue(accounts[1], 2400, { from: owner });

  // account 0 votes for choice 0
  await vote.cast(0, { from: accounts[0] });
  // account 1 votes for choice 1
  await vote.cast(1, { from: accounts[1] });

  assert.equal((await vote.getVoteCount(0)).toNumber(), 1200);
  assert.equal((await vote.getVoteCount(1)).toNumber(), 2400);
});

it("does not let anyone vote twice", async () => {
  await token.issue(accounts[1], 2400, { from: owner });
  await vote.cast(1, { from: accounts[1] });
  await helper.expectError(
    vote.cast(0, { from: accounts[1] }),
    "this user has already placed a vote"
  );

  assert.equal((await vote.getVoteCount(0)).toNumber(), 0);
  assert.equal((await vote.getVoteCount(1)).toNumber(), 2400);
});
```

```
  it("counts number of voters", async () => {
    await token.issue(accounts[0], 1200, { from: owner });
    await token.issue(accounts[1], 2400, { from: owner });
    await token.issue(accounts[2], 3600, { from: owner });

    await vote.cast(0, { from: accounts[0] });
    await vote.cast(1, { from: accounts[1] });
    await vote.cast(2, { from: accounts[2] });

    assert.equal((await vote.voteCount()).toNumber(), 3);
  });

  it("emits a vote cast event", async () => {
    await token.issue(accounts[1], 2400, { from: owner });
    let transaction = await vote.cast(1, { from: accounts[1] });
    assert.equal(transaction.logs[0].event, "VoteCast");
    const eventArguments = transaction.logs[0].args;
    assert.equal(eventArguments.shareholder, accounts[1]);
    assert.equal(eventArguments.choice, 1);
    assert.equal(eventArguments.weight, 2400);
  });

  it("prevents voting for invalid choices", async () => {
    await token.issue(accounts[1], 1200, { from: owner });
    await helper.expectError(
      vote.cast(3, { from: accounts[1] }),
      "there are only 3 choicse to vote on"
    );
  });
});
```

References

[1] The house edge in blackjack - facts you need to know. Available online.

[2] European gaming and betting association. 2017. Available online.

[3] AdStage. Google display ads cpm, cpc, & ctr benchmarks in q1 2018. Available online.

[4] A. G. Association. State of the states 2018 - the aga survey of the commercial casino industry. 2017. Available online.

[5] E. C. Association. Country-by-country report. August 2017. Available online.

[6] E. C. Association. Germany country report. August 2017. Available online.

[7] Cardschat. Rise of the machines: How poker bots infiltrated the online game. Available online.

[8] A. Damodaran. Margins by sector. Available online.

[9] Deloitte. Measuring the economic contribution of the british betting industry. March 2018. Available online.

[10] B. Doctor. Blackjack house edge calculator. Available online.

[11] Ethereum. Ethereum 2.0 specifications. Available at https://github.com/ethereum/eth2.0-specs.

[12] FBI. Manhattan u.s. attorney announces 731 usd million settlement of money laundering and forfeiture complaint with pokerstars and full tilt poker. Jul 2012. Available online.

[13] Go-Globe. The state of mobile gaming industry - statistics and trends [infographic]. Jun 2018. Available online.

[14] J. Kilby, J. Fox, and A. F. Lucas. *Casino operations management*. John Wiley & Sons New York, NY, 2005.

[15] F. on Gambling. House edge percentage chart casino games. Available online.

[16] J. Payton. What's the average bounce rate for a website? Available online.

[17] A. Pratskevich. Instagram ads cpm, cpc, & ctr benchmarks in q1 2018. Available online.

[18] PwC. Pwc research shows industry's economic contributions. June 2017. Available online.

[19] Q. T. Queensland Government Statistician's Office. Australian gambling statistics - 34th edition. October 2018. Available online.

[20] T. Royale. China i-gaming market overview. March 2018. Available online.

[21] Statista. Average weekly time spent playing social casino according to gamers in the united states as of august 2015. Available online.

[22] StrikeSocial. What is a good click-through rate? Available online.

[23] Wikipedia. Provably fair gambling. Available online.

[24] WordStream. Average click-through rate - adwords. Available online.

[25] WordStream. Facebook ad benchmarks. Available online.