Finding Net Credit Utilization Highest Percentage Contributor Choice Algorithm's Proof: Machine Assisted Proof Generation Based Context Engineering

Larry Combs
larry@wlthywise.com

Alex
alex@wlthywise.com

Abstract

This paper details the process of generating a proof of correctness for an algorithm that identifies the optimal credit account to pay off to minimize net credit utilization. The research leverages a Machine-in-the-Loop (MIL) approach, specifically using large language models like Google's Gemini Pro, to assist in the generation of mathematical proofs and corresponding numerical analysis code. We explore the concept of context engineering, where declarative representations of mathematical concepts, such as LaTeX-formatted loop invariants, are used as prompts to guide the Large Language Models in producing desired artifacts. The paper discusses the workflow, from initial human-led proof conditioning to final proof generation, and touches upon related computational problems like the bin-packing problem.

1 Introduction

The problem of managing credit utilization is a common challenge in credit scoring optimization goal based financial management. A key question is how to allocate a payment to have the most significant impact on one's overall credit score. Did leaving all accounts open despite payoff, a given, for the said optimization, or optimizing for account age beyond credit utilization, was also to be find optimized in the process, is also an important consideration of this work. Net credit utilization, is defined as the ratio between total balances and total credit limits proportionally. We explore machine assisted proof as contexts for prompting, however, while concered about less than 0.45 in accuracy against the MATH dataset [1]. The aforementioned evaluation's use of the MATH dataset, shows a dataset like "Metamath Theorem Proving" [2], was not employed to show poor mathematical ability to the degree to which model capability seem to be required for this work.

We chose to isolate away account age optimization, and scope the problem to finding the account which is the highest net credit utilization component in the total percentage, and reduce it without closing any account to further remove accounts which being closed could benefit the consumer with an increase of their average account age. This lends to a bin-packing problem, which we wish to further explore, in future work, however, focusing on the scoped problem for this work, we train ourselves in machine assisted proof based context engineering with Large Language Models [3]. We couldn't make use of an unproven algorithm, as part of financial well-beign related compute, so we set to perform a study, by desiging a process to produce an algorithm and proof for the scoped problem for this work.

2 Process of Machine Assisted Proof Generation

The generation of the final proof and associated code followed a structured, multi-stage process that combined human intuition with the generative power of an Large Language Models. This “Machine Assisted Proof Generation” workflow can be visualized as follows:

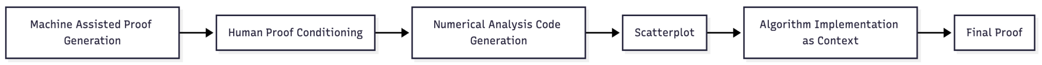
Machine Assisted Proof Generation
Human Proof Conditioning
Numerical Analysis Code Generation
Scatterplot
Algorithm Implementation as Context
Final Proof

Figure 1: Human workflow for machine assisted proof production with implementation

- **Machine Assisted Proof Generation:** Use commercially selected code generation tools from single vendor to generate proof against loop invariants.
- **Human Proof Conditioning:** The process began with a human formulating the initial logical structure of the proof, including defining loop invariants and the overall inductive strategy. These were rough, high-level ideas rather than fully-formed proofs.
- **Numerical Analysis Code Generation:** The conditioned proof fragments, expressed in LaTeX, were used as context in prompts for the Large Language Models to generate Python code for numerical analysis. This step served to validate the mathematical intuition.
- **Scatterplot:** The generated code was used to create visualizations, such as scatterplots, to confirm that the relationships described in the proofs held true over a range of data.
- **Algorithm Implementation as Context:** The Python implementation of the algorithm itself was then used as context for the Large Language Models. In our case, this step was performed by the human researcher, but it is a task that could be automated.
- **Final Proof:** With the validated logic, code, and algorithm structure as context, the Large Language Models was prompted to generate the final, formal proof of correctness in LaTeX.

3 Numerical Analysis and Implementation

A crucial aspect of this research is the acknowledgment of the limitations of standard floating-point arithmetic in financial contexts.

3.1 Notice on Numerical Analysis

The numerical analysis code generated during this process uses IEEE 754 64-bit floating-point numbers. While standard for scientific computing, this representation is susceptible to precision errors, famously leading to issues like the “penny-bug” in financial calculations. For any real-world financial implementation, it is strongly recommended to use decimal-based arithmetic libraries to ensure accuracy. As noted by K. Wellenzohn, “It can be tempting to reach for floating point numbers to represent monetary values...” [4]

3.2 Numerical Analysis Framework Code Generation

The resultant numerical analysis framework was generated to baseline a human written algorithm through a scatter plot.

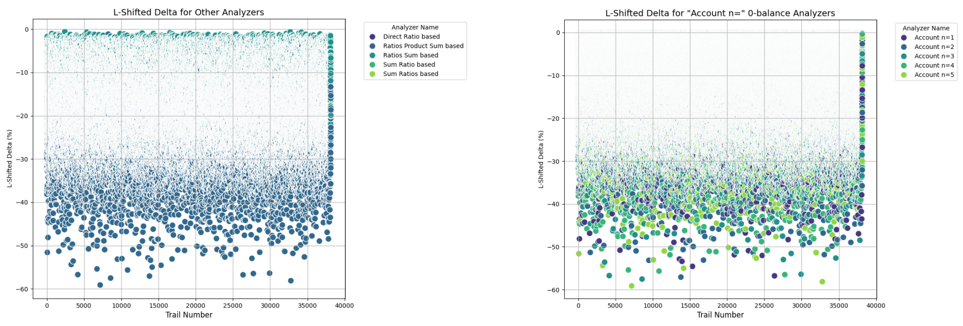

Figure 2: Human written (l) Proof-assisted machine generation (r)

Table 1: Time and Space Complexity of Numerical Analysis Algorithms

Algorithm	**Time Complexity**	**Space Complexity**
`CreditUtilizationSorterDirectRatioLoopBase`	O(n log n)	O(n)
`CreditUtilizationSorterSumRatioLoopInvariantA`	O(n log n)	O(n)
`CreditUtilizationSorterSumRatiosWeightedRatioLoopInvariantB`	O(n log n)	O(n)
`CreditUtilizationSorterRatiosSumWeightedRatioLoopInvariantC`	O(n log n)	O(n)
`CreditUtilizationSorterRatiosProductSumLoopInvariantD`	O(n log n)	O(n)
`best_fit_bin_packing_simulate_payoff_utilization_percentage_delta_for_account_n`	O(n^2)	O(n)

4 Context Engineering and Proof Design

A core theme of this work is "context engineering", of algorithms where the structure and content of the prompt provided to the Large Language Models are carefully designed to elicit a specific, structured output. We found that providing declarative context, such as a LaTeX proof fragment, was highly effective in guiding the model. This work makes use of machine assisted proof generation [5]. The proof based code generation approach is distinct to Self-planning Code Generation with Large Language Models [6].

4.1 Meta Prompts

The following are examples of the meta-prompts [7] used. The {LaTeX_CONTEXT_HERE} placeholder was replaced with the content of one of the intermediate proof files, when performing Large Language Models based code generation.

```
{LaTeX_CONTEXT_HERE}
Given the above given create a system which sorts the inputs of the proof,
    as method arguments of set of tuples of balance and credit limit. Use
   the loop invariant to sort and not the terminating state.
```

```
{LaTeX_CONTEXT_HERE}
Given the above taken create a system which sorts the inputs of the proof,
    as method arguments of set of tuples of balance and credit limit.
```

```
{LaTeX_CONTEXT_HERE}
Given the above taken create a system which sorts the inputs of the proof,
    as method arguments of set of tuples of balance and credit limit.
```

```
{LaTeX_CONTEXT_HERE}
Please take the above proof of correctness and replace the loop invariant with the following:

(bk _ sigma (bi/li))/lk > (b(k+1) _ sigma (bi/li))/l(k+1)

write a new proof with the above loop invariant to find a new terminating condition
```

```
{LaTeX_CONTEXT_HERE}
Please take the above proof of correctness and replace the loop invariant with the following:

(bk _ sigma (bi/li))/lk > (b(k+1) _ sigma (bi/li))/l(k+1)

write a new proof with the above loop invariant to find a new terminating condition
```

```
{LaTeX_CONTEXT_HERE}
Transpile the attached derivation of a mathematical sum in to Latex
```

4.2 Proofs as Context Prompts

The following LaTeX snippets represent the intermediate, exploratory proofs that were used as context in the prompts. The actual source of these context prompt fragments is being released through Github with the research disclosure of this work.

4.2.1 Loop Invariant A

```
\documentclass{article}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage[left=1in, right=1in, top=1in, bottom=1in]{geometry}

% Replaced the old 'pseudocode' package with the modern 'algorithm'
% and 'algpseudocode' packages, which support the syntax being used.
\usepackage{algorithm}
\usepackage{algpseudocode}

\title{Proof of Correctness for Net Credit Limit Utilization Minimization Algorithm}
\author{REMOVED_FOR_PRIVACY_REASONS}
\date{}

\begin{document}

\maketitle

\section{Problem Definition}

We are given a set of credit card accounts, which can be defined as follows:
```

```latex
\begin{itemize}
    \item $L = \langle l_1, \dots, l_{n} \rangle$ is a sequence of credit limits.
    \item $b = \langle b_1, \dots, b_{n} \rangle$ is a sequence of corresponding credit card balances.
    \item $C = \{(b_1, l_1), \dots, (b_{n}, l_{n})\}$ is a set of tuples, where each tuple represents a single credit card account.
    \item $n$ is the total number of credit card accounts. We assume $n > 0$.
    \item $k$ is the loop index, where $0 < k < n$.
\end{itemize}

We want to prove the correctness of an algorithm that iterates through the credit card accounts, which are pre-sorted by their balance in ascending order.

\section{Pseudocode}

The algorithm first sorts the set of credit card accounts $C$ based on $ \frac{b_{i}}{\sum_{i=1}^{n} L_i} $ and then iterates from the first credit card account to the second-to-last credit card account in the sorted order.

% Switched to the 'algorithm' and 'algorithmic' environments.
% Updated the syntax to match the 'algpseudocode' package.
% This includes using \Procedure, \State, \If, \For, \gets, \Call, and \Comment.
% Also ensured all mathematical expressions are in math mode ($...$).
\begin{algorithm}
\caption{ProcessSortedBalances($C$)}
\begin{algorithmic}[1]
\Procedure{ProcessSortedBalances}{$C$}
    \Comment{Input: A set of credit card accounts tuples $C = \{(b_i, l_i)\}$}
    \Comment{Output: Processes credit card accounts in order of increasing balance.}
    \State $S \gets \Call{SortByBalance}{C}$ \Comment{Sort $C$ by balance $b_i$ in ascending order}
    \State $n \gets \Call{Length}{S}$
    \If{$n > 0$}
        \For{$k \gets 1$ to $n-1$}
            \State \Comment{Loop body. Operations on $S_k$ and $S_{k+1}$ could be performed here.}
            \State \Comment{The loop invariant holds at the beginning of each iteration.}
        \EndFor
    \EndIf
\EndProcedure
\end{algorithmic}
\end{algorithm}

\section{Loop Invariant Proof}

```

```
We will prove the correctness using the following loop invariant. The invariant is evaluated at the start of each iteration of the \textbf{FOR} loop for index $k$.

\subsection{The Invariant}

\textbf{Loop Invariant:} At the start of the iteration for a given index $k$ (where $0 < k < n$), the following inequality holds:
$$ \frac{b_{k}}{\sum_{i=1}^{n} L_i} < \frac{b_{k+1}}{\sum_{i=1}^{n} L_i} $$
where $b_k$ and $b_{k+1}$ are the balances of the credit card accounts at indices $k$ and $k+1$ in the sorted sequence $S$. Since the denominator $\sum_{i=1}^{n} L_i$ is a positive constant (assuming credit limits are positive), this invariant is equivalent to stating that $b_k < b_{k+1}$.

\subsection{Proof by Induction}

\subsubsection{Initialization}
We must show that the invariant is true before the first loop iteration.
The loop starts with $k=1$. The invariant for this case is:
$$ \frac{b_{1}}{\sum_{i=1}^{n} L_i} < \frac{b_{2}}{\sum_{i=1}^{n} L_i} $$
This simplifies to $\frac{b_{2}}{\sum_{i=1}^{n} L_i} < \frac{b_{3}}{\sum_{i=1}^{n} L_i}$.

The pseudocode's first step is to sort the sequence of credit card accounts $S$ by the ratio of the balance and the sum of the credit limits in ascending order. By the definition of this sorting operation, for any two adjacent elements $S_i$ and $S_{i+1}$, the ratio of the balance and the sum of the credit limits by $S_i$ must be less than or equal to the ratio of the balance and the sum of the credit limits by $S_{i+1}$. Assuming no two credit card accounts have the exact same ratio of the balance and the sum of the credit limits, this means $\frac{b_{1}}{\sum_{i=1}^{n} L_i} < \frac{b_{2}}{\sum_{i=1}^{n} L_i}$. Therefore, the invariant holds for the initial case $k=1$.

\subsubsection{Maintenance}
We assume the loop invariant holds for an arbitrary iteration $k$, and we must show that it holds for the next iteration, $k+1$. The loop runs as long as $k < n-1$. Let's assume the invariant holds for an iteration $j$, where $0 < j < n-1$.

\textbf{Assumption:}
$$ \frac{b_{j}}{\sum_{i=1}^{n} L_i} < \frac{b_{j+1}}{\sum_{i=1}^{n} L_i} \quad (\text{which implies } b_j < b_{j+1}) $$

We need to prove that at the start of the next iteration, where the index is $j+1$, the invariant still holds. The invariant for index $j+1$ is:
$$ \frac{b_{j+1}}{\sum_{i=1}^{n} L_i} < \frac{b_{j+2}}{\sum_{i=1}^{n} L_i} $$
This inequality is equivalent to $b_{j+1} < b_{j+2}$.

Just as in the initialization step, this condition is guaranteed to be true because the sequence $S$ was sorted by the ratio of the balance
```

```
    and the sum of the credit limits as a precondition. The loop does not
    modify the order of elements in $S$. Thus, for any valid index $j+1$,
    it must be that $b_{j+1} < b_{j+2}$. The invariant is therefore
    maintained through each iteration of the loop.

\subsubsection{Termination}
The loop terminates when $k = n-1$. The final iteration of the loop body
    is for $k = n-1$.

At the start of this final iteration, the maintenance property guarantees
    that the invariant holds for $k = n-1$, which is:
$$ \frac{b_{n-2}}{\sum_{i=1}^{n} L_i} < \frac{b_{n-1}}{\sum_{i=1}^{n} L_i}
     $$
This shows that $b_{n-2} < b_{n-1}$.

When the loop terminates, we have successfully shown that the invariant
    $b_k < b_{k+1}$ held for all $k$ from $0$ to $n-2$. This means that
    $b_0 < b_1 < b_2 < \dots < b_{n-1}$. This confirms that the algorithm
    correctly maintains the property of strictly increasing balances
    throughout its execution, which is what we set out to prove.

\end{document}
```

4.2.2 Loop Invariant B

```
\documentclass{article}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage[left=1in, right=1in, top=1in, bottom=1in]{geometry}

% Using modern 'algorithm' and 'algpseudocode' packages
\usepackage{algorithm}
\usepackage{algpseudocode}
\usepackage{amsthm}

\title{Proof of Correctness for Finding the Largest Credit Utilization
    Ratio}
\author{REMOVED_FOR_PRIVACY_REASONS}
\date{}

\begin{document}

\maketitle

\section{Problem Definition}

We are given a set of credit card accounts, which can be defined as
    follows:
\begin{itemize}
    \item $L = \langle l_1, \dots, l_{n} \rangle$ is a sequence of credit
        limits.
    \item $b = \langle b_1, \dots, b_{n}) \rangle$ is a sequence of
        corresponding credit card balances.
    \item $C = \{(b_1, l_1), \dots, (b_{n}, l_{n})\}$ is a set of tuples,
        where each tuple represents a single credit card account. We assume
         $l_i > 0$ for all $i > 0$.
    \item $n$ is the total number of credit card accounts. We assume $n >
        1$.
\end{itemize}

The goal is to prove the correctness of an algorithm that finds and
    returns the single account tuple $(b_i, l_i)$ from $C$ with the **
    largest** credit utilization ratio $r_i = b_i / l_i$. The algorithm
    uses the following sorting key, $R_i$, which is equivalent for sorting
    purposes:
$$ R_i = \frac{b_{i}\sum_{j=1}^{n} l_j}{l_{i}\sum_{j=1}^{n} b_j} $$

\section{Pseudocode}

The algorithm first sorts the set of accounts $C$ based on the key $R_i$
    in **descending** order. Because the list is sorted this way, the
    account with the maximum utilization ratio is simply the first element
    of the sorted list.

\begin{algorithm}
\caption{FindLargestUtilizationAccount($C$)}
\begin{algorithmic}[1]
\Procedure{FindLargestUtilizationAccount}{$C$}
    \Comment{Input: A set of credit card account tuples $C = \{(b_i, l_i)
        \}$}
    \Comment{Output: The tuple $(b_k, l_k)$ with the largest utilization
        ratio $b_k/l_k$.}
    \State $L_{total} \gets \sum_{i=1}^{n} l_i$
    \State $B_{total} \gets \sum_{i=1}^{n} b_i$
    \State $S \gets \Call{Sort}{C}$ \Comment{Sort $C$ by $R_i = \frac{b_{i
        } \cdot L_{total}}{l_{i} \cdot B_{total}}$ in **descending** order}
    \State $n \gets \Call{Length}{S}$
    \If{$n > 1$}
        \For{$k \gets 1$ to $n-1$}
            \State \Comment{This loop's purpose is for the proof of
                correctness, verifying S is sorted.}
        \EndFor
    \EndIf
    \State \Comment{After sorting, the first element $S_1$ is the account
        with the largest ratio.}
    \State \Return $S_1$
\EndProcedure
\end{algorithmic}
\end{algorithm}

\section{Loop Invariant Proof}

We will prove the correctness of the sorting procedure using a loop
    invariant. The invariant is evaluated at the start of each iteration of
     the \textbf{FOR} loop for index $k$.

\subsection{The Invariant}
```

```

\textbf{Loop Invariant:} At the start of the iteration for a given index $k$ (where $1 \le k < n$), the following inequality holds for the elements at indices $k$ and $k+1$ in the sorted sequence $S$:
$$ \frac{b_{k}\sum_{i=1}^{n} l_i}{l_{k}\sum_{i=1}^{n} b_i} > \frac{b_{k+1}\sum_{i=1}^{n} l_i}{l_{k+1}\sum_{i=1}^{n} b_i} $$
Since the total limit, $\sum_{i=1}^{n} l_i$, and the total balance, $\sum_{i=1}^{n} b_i$, are positive constants for all elements, this invariant is equivalent to stating that the utilization ratio of the $k$-th element is **greater than** that of the $(k+1)$-th element: $b_k/l_k > b_{k+1}/l_{k+1}$.

\subsection{Proof by Induction}

\subsubsection{Initialization}
We must show that the invariant is true before the first loop iteration. The loop starts with $k=1$. The invariant for this case is:
$$ \frac{b_{1}\sum_{i=1}^{n} l_i}{l_{1}\sum_{i=1}^{n} b_i} > \frac{b_{2}\sum_{i=1}^{n} l_i}{l_{2}\sum_{i=1}^{n} b_i} $$
The algorithm's first active step is to sort the sequence of accounts $S$ by the key $R_i$ in **descending** order. By the definition of this sorting operation, for any two adjacent elements $S_j$ and $S_{j+1}$, the sorting key of $S_j$ must be greater than or equal to the sorting key of $S_{j+1}$. Assuming no two accounts have the exact same utilization ratio, this means the key for the first element, $S_1$, is strictly **greater than** the key for the second element, $S_2$. Therefore, the invariant holds for the initial case $k=1$.

\subsubsection{Maintenance}
We assume the loop invariant holds for an arbitrary iteration $k$ (where $1 \le k < n-1$), and we must show that it also holds for the next iteration, $k+1$.

\textbf{Inductive Hypothesis:} Assume at the start of iteration $k$, the following is true:
$$ \frac{b_{k}\sum_{i=1}^{n} l_i}{l_{k}\sum_{i=1}^{n} b_i} > \frac{b_{k+1}\sum_{i=1}^{n} l_i}{l_{k+1}\sum_{i=1}^{n} b_i} $$
We need to prove that at the start of the next iteration, where the index is $k+1$, the invariant still holds. The invariant for index $k+1$ is:
$$ \frac{b_{k+1}\sum_{i=1}^{n} l_i}{l_{k+1}\sum_{i=1}^{n} b_i} > \frac{b_{k+2}\sum_{i=1}^{n} l_i}{l_{k+2}\sum_{i=1}^{n} b_i} $$
This inequality is guaranteed to be true for the same reason as in the initialization step. The sequence $S$ was sorted by the key $R_i$ as a precondition before the loop began. The loop itself does not modify the order of elements in $S$. Thus, the sorted property holds for any pair of adjacent elements in the sequence, including $S_{k+1}$ and $S_{k+2}$. The invariant is therefore maintained through each iteration of the loop.

\subsubsection{Termination}
The loop terminates when $k$ becomes $n$. The final iteration of the loop body is for $k = n-1$.

```

```
At the start of this final iteration, the maintenance property guarantees
    that the invariant holds for $k = n-1$, which is:
$$ \frac{b_{n-1}\sum_{i=1}^{n} l_i}{l_{n-1}\sum_{i=1}^{n} b_i} > \frac{b_{
    n}\sum_{i=1}^{n} l_i}{l_{n}\sum_{i=1}^{n} b_i} $$
This is equivalent to showing $b_{n-1}/l_{n-1} > b_{n}/l_{n}$.

When the loop terminates, the invariant has held true for all values of
    $k$ from $1$ to $n-1$. This implies that for any adjacent pair of
    accounts $(S_k, S_{k+1})$ in the sequence, the inequality $b_k/l_k > b_
    {k+1}/l_{k+1}$ is true. This establishes a chain of inequalities for
    the entire sequence:
$$ \frac{b_1}{l_1} > \frac{b_2}{l_2} > \dots > \frac{b_{n-1}}{l_{n-1}} > \
    frac{b_n}{l_n} $$
This confirms that the algorithm correctly sorts the accounts in **
    descending** order of their credit utilization ratio. Consequently, the
     algorithm correctly identifies the account with the **maximum**
    utilization ratio by returning the first element of this sorted
    sequence, $S_1$.

\qedsymbol

\end{document}
```

4.2.3 Loop Invariant C

```
\documentclass{article}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage[left=1in, right=1in, top=1in, bottom=1in]{geometry}

% Using modern 'algorithm' and 'algpseudocode' packages
\usepackage{algorithm}
\usepackage{algpseudocode}
\usepackage{amsthm}

\title{Revised Proof of Correctness for Finding the Largest Credit
    Utilization Ratio}
\author{REMOVED_FOR_PRIVACY_REASONS}
\date{09/14/2025}

\begin{document}

\maketitle

\section{Problem Definition}

We are given a set of credit card accounts, which can be defined as
    follows:
\begin{itemize}
    \item $L = \langle l_1, \dots, l_{n} \rangle$ is a sequence of credit
        limits.
    \item $b = \langle  b_1, \dots, b_{n} \rangle$ is a sequence of
        corresponding credit card balances.
    \item $C = \{(b_1, l_1), \dots, (b_{n}, l_{n})\}$ is a set of tuples, where each tuple represents a single credit card account. We assume $l_i > 0$ for all $i > 0$.
    \item $n$ is the total number of credit card accounts. We assume $n > 1$.
\end{itemize}

The goal is to prove the correctness of an algorithm that finds and returns the single account tuple $(b_i, l_i)$ from $C$ with the **largest** credit utilization ratio $r_i = b_i / l_i$. The algorithm uses the following sorting key, $R'_i$:
$$ R'_i = \frac{b_{i}\sum_{j=1}^{n} (b_j/l_j)}{l_{i}} $$

\section{Pseudocode}

The algorithm first sorts the set of accounts $C$ based on the key $R'_i$ in **descending** order. Because the list is sorted this way, the account with the maximum utilization ratio is simply the first element of the sorted list.

\begin{algorithm}
\caption{FindLargestUtilizationAccount($C$)}
\begin{algorithmic}[1]
\Procedure{FindLargestUtilizationAccount}{$C$}
    \Comment{Input: A set of credit card account tuples $C = \{(b_i, l_i)\}$}
    \Comment{Output: The tuple $(b_k, l_k)$ with the largest utilization ratio $b_k/l_k$.}
    \State $R_{normalizer} \gets \sum_{i=1}^{n} (b_i/l_i)$ \Comment{Sum of all individual utilization ratios}
    \State $S \gets \Call{Sort}{C}$ \Comment{Sort $C$ by $R'_i = \frac{b_{i} \cdot R_{normalizer}}{l_{i}}$ in **descending** order}
    \State $n \gets \Call{Length}{S}$
    \If{$n > 1$}
        \For{$k \gets 1$ to $n-1$}
            \State \Comment{This loop's purpose is for the proof of correctness, verifying S is sorted.}
        \EndFor
    \EndIf
    \State \Comment{After sorting, the first element $S_1$ is the account with the largest ratio.}
    \State \Return $S_1$
\EndProcedure
\end{algorithmic}
\end{algorithm}

\section{Loop Invariant Proof}

We will prove the correctness of the sorting procedure using a loop invariant. The invariant is evaluated at the start of each iteration of the \textbf{FOR} loop for index $k$.

\subsection{The Invariant}

```

```
\textbf{Loop Invariant:} At the start of the iteration for a given index
    $k$ (where $1 \le k < n$), the following inequality holds for the
    elements at indices $k$ and $k+1$ in the sorted sequence $S$:
$$ \frac{b_{k}\sum_{i=1}^{n} (b_i/l_i)}{l_{k}} > \frac{b_{k+1}\sum_{i=1}^{
    n} (b_i/l_i)}{l_{k+1}} $$
Let $r_i = b_i/l_i$ be the utilization ratio for account $i$. The term $\
    sum_{i=1}^{n} (b_i/l_i)$ is the sum of all individual utilization
    ratios. Let's call this sum $R_{sum} = \sum_{i=1}^{n} r_i$. Since all
    $l_i > 0$ and we can assume at least one $b_i \ge 0$ to avoid a trivial
     case, $R_{sum}$ is a non-negative constant. Assuming $R_{sum} > 0$, we
     can divide both sides of the invariant by this constant, simplifying
    the invariant to:
$$ \frac{b_k}{l_k} > \frac{b_{k+1}}{l_{k+1}} $$
This demonstrates that the sorting key $R'_i$ is equivalent to the actual
    utilization ratio $r_i$ for sorting purposes.

\subsection{Proof by Induction}

\subsubsection{Initialization}
We must show that the invariant is true before the first loop iteration.
    The loop starts with $k=1$. The invariant for this case is:
$$ \frac{b_{1}\sum_{i=1}^{n} (b_i/l_i)}{l_{1}} > \frac{b_{2}\sum_{i=1}^{n}
     (b_i/l_i)}{l_{2}} $$
The algorithm's first active step is to sort the sequence of accounts $S$
    by the key $R'_i$ in **descending** order. By the definition of this
    sorting operation, for any two adjacent elements $S_j$ and $S_{j+1}$,
    the sorting key of $S_j$ must be greater than or equal to the sorting
    key of $S_{j+1}$. Assuming no two accounts have the exact same
    utilization ratio, the key for the first element, $S_1$, is strictly **
    greater than** the key for the second element, $S_2$. Therefore, the
    invariant holds for the initial case $k=1$.

\subsubsection{Maintenance}
We assume the loop invariant holds for an arbitrary iteration $k$ (where
    $1 \le k < n-1$), and we must show that it also holds for the next
    iteration, $k+1$.

\textbf{Inductive Hypothesis:} Assume at the start of iteration $k$, the
    following is true:
$$ \frac{b_{k}\sum_{i=1}^{n} (b_i/l_i)}{l_{k}} > \frac{b_{k+1}\sum_{i=1}^{
    n} (b_i/l_i)}{l_{k+1}} $$
We need to prove that at the start of the next iteration, where the index
    is $k+1$, the invariant still holds. The invariant for index $k+1$ is:
$$ \frac{b_{k+1}\sum_{i=1}^{n} (b_i/l_i)}{l_{k+1}} > \frac{b_{k+2}\sum_{i
    =1}^{n} (b_i/l_i)}{l_{k+2}} $$
This inequality is guaranteed to be true for the same reason as in the
    initialization step. The sequence $S$ was sorted by the key $R'_i$ as a
     precondition before the loop began. The loop itself does not modify
    the order of elements in $S$. Thus, the sorted property holds for any
    pair of adjacent elements in the sequence, including $S_{k+1}$ and $S_{
    k+2}$. The invariant is therefore maintained through each iteration of
    the loop.

\subsubsection{Termination}
```

```
The loop terminates when $k$ becomes $n$. The final iteration of the loop
    body is for $k = n-1$.

At the start of this final iteration, the maintenance property guarantees
    that the invariant holds for $k = n-1$, which is:
$$ \frac{b_{n-1}\sum_{i=1}^{n} (b_i/l_i)}{l_{n-1}} > \frac{b_{n}\sum_{i
    =1}^{n} (b_i/l_i)}{l_{n}} $$
This is equivalent to showing $b_{n-1}/l_{n-1} > b_{n}/l_{n}$.

When the loop terminates, the invariant has held true for all values of
    $k$ from $1$ to $n-1$. This establishes a transitive chain of
    inequalities for the entire sequence based on the simplified invariant:
$$ \frac{b_1}{l_1} > \frac{b_2}{l_2} > \dots > \frac{b_{n-1}}{l_{n-1}} > \
    frac{b_n}{l_n} $$
This chain of inequalities is the **terminating condition** that holds
    true once the loop has finished. It confirms that the algorithm
    correctly sorts the accounts in **descending** order of their credit
    utilization ratio. Consequently, the algorithm correctly identifies the
     account with the **maximum** utilization ratio, which must be the
    first element of this sorted sequence, $S_1$.

\qedsymbol

\end{document}
```

4.2.4 Loop Invariant D

```
\documentclass{article}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage[left=1in, right=1in, top=1in, bottom=1in]{geometry}

% Using modern 'algorithm' and 'algpseudocode' packages
\usepackage{algorithm}
\usepackage{algpseudocode}
\usepackage{amsthm}

\title{Proof of Correctness for Finding the Largest Balance-Limit Product}
\author{REMOVED_FOR_PRIVACY_REASONS}
\date{09/14/2025}

\begin{document}

\maketitle

\section{Problem Definition}

We are given a set of credit card accounts, which can be defined as
    follows:
\begin{itemize}
    \item $L = \langle l_1, \dots, l_{n} \rangle$ is a sequence of credit
        limits.
    \item $b = \langle  b_1, \dots, b_{n} \rangle$ is a sequence of corresponding credit card balances.
    \item $C = \{(b_1, l_1), \dots, (b_{n}, l_{n})\}$ is a set of tuples, where each tuple represents a single credit card account. We assume $l_i > 0$ for all $i > 0$.
    \item $n$ is the total number of credit card accounts. We assume $n > 1$.
\end{itemize}

The goal is to prove the correctness of an algorithm that finds and returns the single account tuple $(b_i, l_i)$ from $C$ with the **largest** balance-limit product $p_i = b_i \cdot l_i$. The algorithm uses the following sorting key, $R_i$:
$$ R_i = \frac{b_{i}l_{i}}{\sum_{j=1}^{n} b_j l_j} $$

\section{Pseudocode}

The algorithm first sorts the set of accounts $C$ based on the key $R_i$ in **descending** order. Because the list is sorted this way, the account with the maximum balance-limit product is simply the first element of the sorted list.

\begin{algorithm}
\caption{FindLargestBalanceLimitProductAccount($C$)}
\begin{algorithmic}[1]
\Procedure{FindLargestBalanceLimitProductAccount}{$C$}
    \Comment{Input: A set of credit card account tuples $C = \{(b_i, l_i)\}$}
    \Comment{Output: The tuple $(b_k, l_k)$ with the largest balance-limit product $b_k \cdot l_k$.}
    \State $BL_{sum} \gets \sum_{i=1}^{n} b_i l_i$ \Comment{Sum of all individual balance-limit products}
    \State $S \gets \Call{Sort}{C}$ \Comment{Sort $C$ by $R_i = \frac{b_{i}l_{i}}{BL_{sum}}$ in **descending** order}
    \State $n \gets \Call{Length}{S}$
    \If{$n > 1$}
        \For{$k \gets 1$ to $n-1$}
            \State \Comment{This loop's purpose is for the proof of correctness, verifying S is sorted.}
        \EndFor
    \EndIf
    \State \Comment{After sorting, the first element $S_1$ is the account with the largest product.}
    \State \Return $S_1$
\EndProcedure
\end{algorithmic}
\end{algorithm}

\section{Loop Invariant Proof}

We will prove the correctness of the sorting procedure using a loop invariant. The invariant is evaluated at the start of each iteration of the \textbf{FOR} loop for index $k$.

```

```latex
\subsection{The Invariant}

\textbf{Loop Invariant:} At the start of the iteration for a given index
    $k$ (where $1 \le k < n$), the following inequality holds for the
    elements at indices $k$ and $k+1$ in the sorted sequence $S$:
$$ \frac{b_{k}l_{k}}{\sum_{i=1}^{n} b_i l_i} > \frac{b_{k+1}l_{k+1}}{\sum_
    {i=1}^{n} b_i l_i} $$
Let $BL_{sum} = \sum_{i=1}^{n} b_i l_i$. Since all $l_i > 0$ and we can
    assume at least one $b_i \ge 0$ to avoid a trivial case, $BL_{sum}$ is
    a non-negative constant. Assuming $BL_{sum} > 0$, we can multiply both
    sides of the invariant by this constant, simplifying the invariant to:
$$ b_k l_k > b_{k+1} l_{k+1} $$
This demonstrates that the sorting key $R_i$ is equivalent to the balance-
    limit product $p_i = b_i l_i$ for sorting purposes.

\subsection{Proof by Induction}

\subsubsection{Initialization}
We must show that the invariant is true before the first loop iteration.
    The loop starts with $k=1$. The invariant for this case is:
$$ \frac{b_{1}l_{1}}{\sum_{i=1}^{n} b_i l_i} > \frac{b_{2}l_{2}}{\sum_{i
    =1}^{n} b_i l_i} $$
The algorithm's first active step is to sort the sequence of accounts $S$
    by the key $R_i$ in **descending** order. By the definition of this
    sorting operation, for any two adjacent elements $S_j$ and $S_{j+1}$,
    the sorting key of $S_j$ must be greater than or equal to the sorting
    key of $S_{j+1}$. Assuming no two accounts have the exact same balance-
    limit product, the key for the first element, $S_1$, is strictly **
    greater than** the key for the second element, $S_2$. Therefore, the
    invariant holds for the initial case $k=1$.

\subsubsection{Maintenance}
We assume the loop invariant holds for an arbitrary iteration $k$ (where
    $1 \le k < n-1$), and we must show that it also holds for the next
    iteration, $k+1$.

\textbf{Inductive Hypothesis:} Assume at the start of iteration $k$, the
    following is true:
$$ \frac{b_{k}l_{k}}{\sum_{i=1}^{n} b_i l_i} > \frac{b_{k+1}l_{k+1}}{\sum_
    {i=1}^{n} b_i l_i} $$
We need to prove that at the start of the next iteration, where the index
    is $k+1$, the invariant still holds. The invariant for index $k+1$ is:
$$ \frac{b_{k+1}l_{k+1}}{\sum_{i=1}^{n} b_i l_i} > \frac{b_{k+2}l_{k+2}}{\
    sum_{i=1}^{n} b_i l_i} $$
This inequality is guaranteed to be true for the same reason as in the
    initialization step. The sequence $S$ was sorted by the key $R_i$ as a
    precondition before the loop began. The loop itself does not modify the
     order of elements in $S$. Thus, the sorted property holds for any pair
     of adjacent elements in the sequence, including $S_{k+1}$ and $S_{k+2}
    $. The invariant is therefore maintained through each iteration of the
    loop.

\subsubsection{Termination}
```

```
The loop terminates when $k$ becomes $n$. The final iteration of the loop body is for $k = n-1$.

At the start of this final iteration, the maintenance property guarantees that the invariant holds for $k = n-1$, which is:
$$ \frac{b_{n-1}l_{n-1}}{\sum_{i=1}^{n} b_i l_i} > \frac{b_{n}l_{n}}{\sum_{i=1}^{n} b_i l_i} $$
This is equivalent to showing $b_{n-1}l_{n-1} > b_{n}l_{n}$.

When the loop terminates, the invariant has held true for all values of $k$ from $1$ to $n-1$. This establishes a transitive chain of inequalities for the entire sequence based on the simplified invariant:
$$ b_1 l_1 > b_2 l_2 > \dots > b_{n-1}l_{n-1} > b_n l_n $$
This chain of inequalities is the **terminating condition** that holds true once the loop has finished. It confirms that the algorithm correctly sorts the accounts in **descending** order of their balance-limit product. Consequently, the algorithm correctly identifies the account with the **maximum** balance-limit product, which must be the first element of this sorted sequence, $S_1$.

\qedsymbol

\end{document}
```

5 Final Machine Generated Proof

The culmination of the process is the final proof of correctness, generated by the Large Language Models. This proof uses a direct argument rather than a loop invariant, as the core logic is straightforward.

5.1 Problem Definition

We are given a set of credit card accounts, which can be defined as follows:

- $C = \{(b_1, l_1), \ldots, (b_n, l_n)\}$ is a set of tuples, where each tuple represents a single credit card account with balance b_i and limit l_i.
- We assume $l_i > 0$ for all i, and $n > 0$.
- The total balance is $B_{total} = \sum_{i=1}^{n} b_i$.
- The total limit is $L_{total} = \sum_{i=1}^{n} l_i$.
- The initial net credit utilization is $U_{initial} = \frac{B_{total}}{L_{total}}$.

The objective is to find the single account k which, if its balance b_k were set to zero, would result in the minimum possible net credit utilization. Let U'_k be the new utilization after paying off account k.

$$U'_k = \frac{B_{total} - b_k}{L_{total}}$$

The goal is to prove the correctness of an algorithm that finds the account k that minimizes U'_k.

5.2 Algorithm

The most efficient algorithm to achieve this goal is to identify the account with the highest balance. This can be done by sorting the accounts by balance in descending order and selecting the first element.

Algorithm 1 FindOptimalAccountToPayoff(C)

```
1: procedure FindOptimalAccountToPayoff($C$)    ▷ Input: A set of credit card account tuples $C = \{(b_i, l_i)\}$    ▷ Output: The tuple $(b_k, l_k)$ that minimizes the net utilization if paid off.
2:     $S \leftarrow$ SortByBalanceDescending($C$)
3:     return $S_1$    ▷ Return the first element, which has the highest balance.
4: end procedure
```

5.3 Proof of Correctness

We will use a direct proof to show that selecting the account with the maximum balance correctly minimizes the resulting net credit utilization. A loop invariant is not necessary here because the correctness of the algorithm relies on a direct mathematical relationship rather than a step-by-step state change.

5.3.1 Argument

Let us consider any two distinct accounts from the set C, account j and account k, with balances b_j and b_k respectively. Let's assume, without loss of generality, that account j has a higher balance than account k.

$$b_j > b_k$$

Let U'_j be the resulting net utilization if account j is paid off, and U'_k be the resulting net utilization if account k is paid off.

$$U'_j = \frac{B_{total} - b_j}{L_{total}}$$

$$U'_k = \frac{B_{total} - b_k}{L_{total}}$$

Our goal is to determine which of these two resulting utilization figures is smaller. We start with our assumption:

$$b_j > b_k$$

Multiplying by -1 reverses the inequality:

$$-b_j < -b_k$$

Adding the constant B_{total} to both sides does not change the inequality:

$$B_{total} - b_j < B_{total} - b_k$$

Since we assume that credit limits are positive, the total limit $L_{total} = \sum_{i=1}^{n} l_i$ must be a positive constant. Therefore, we can divide both sides by L_{total} without changing the direction of the inequality:

$$\frac{B_{total} - b_j}{L_{total}} < \frac{B_{total} - b_k}{L_{total}}$$

By substitution with our definitions of U'_j and U'_k, we get:

$$U'_j < U'_k$$

5.3.2 Conclusion

The result $U'_j < U'_k$ demonstrates that for any two accounts, paying off the account with the higher balance (account j) results in a lower final net credit utilization than paying off the account with the lower balance (account k).

By logical extension, this principle applies to the entire set of accounts. The account with the single **highest** balance will yield a lower resulting net utilization than any other account in the set.

Therefore, the algorithm that identifies the account with the maximum balance is correct in its approach to finding the single account that will minimize the net credit utilization percentage when paid off. The 'FindOptimalAccountToPayoff' procedure, which sorts the accounts by balance in descending order and returns the first element, correctly solves the problem.

6 Future Work and Relation to Bin Packing

Beyond the use of Snowball and Avalanche [8], we seek to optimize for average account age, while employing the findings of this paper in to our work.

Future research could explore this connection further. The constraints of minimizing utilization while adhering to a total payment budget could be modeled as a variation of the bin packing or knapsack problem. The works of Shi et al. on "A Survey on Large Language Models for Mathematical Reasoning" [9] and Collins et al. on "Machine Learning for the Multi-Dimensional Bin Packing Problem: Literature Review and Empirical Evaluation" [10] provide potential frameworks for tackling such problems, possibly using Large Language Models to explore the solution space or even generate novel approximation algorithms [10].

7 Conclusion

This paper has demonstrated a workflow for machine-assisted proof generation, leveraging large language models and context engineering to produce formal proofs and numerical analysis code from high-level conceptual prompts. We successfully generated a proof of correctness for an algorithm that minimizes net credit utilization by paying off the highest balance account. This work highlights the potential of Large Language Models as powerful assistants in the formal verification and exploration of algorithms, while also acknowledging the need for human oversight and awareness of limitations, such as those in floating-point arithmetic.

References

[1] Roy, S. T., Baral, A., Jhaveri A., Baig Y., "Can LLMs understand Math? Exploring the Pitfalls in Mathematical Reasoning", arXiv:2505.15623v1.

[2] "MATH Dataset Comparison", `https://github.com/hendrycks/math/blob/main/dataset_comparison.png`.

[3] Mei, L. et al., "A Survey of Context Engineering for Large Language Models", arXiv:2507.13334.

[4] Wellenzohn, K., "Why using floats for money is a bad idea", `https://k13n.io/posts/20250412_floats/`.

[5] Tao, T., "Machine Assisted Proof". `https://www.ams.org/notices/202501/rnoti-p6.pdf`

[6] Jiang, X. et al., "Self-planning Code Generation with Large Language Models", arXiv:2303.06689.

[7] "Meta Prompting for AI Systems", arXiv:2311.11482v7.

[8] McAllister, E. (2018). *A snowball's chance: Debt snowball vs. debt avalanche.* James Madison University.

[9] Shi, Z., Yu, L., Madaan, A., and Yang, L. (2024). "A Survey on Large Language Models for Mathematical Reasoning", arXiv:2506.08446.

[10] Collins, J., School, T. G. B., & Box, G. (2023). "Machine Learning for the Multi-Dimensional Bin Packing Problem: Literature Review and Empirical Evaluation", arXiv:2312.08103.