Formalisation of Davenport–Schinzel Sequences and their Estimates in Lean 4

Antoine du Fresne von Hohenesche

January 15, 2026

Abstract

This document presents the formalisation of Davenport–Schinzel sequences in Lean 4. It covers the foundational definitions, the equivalence between propositional and some computable definitions, exact values for the associated extremal function $\lambda_s(n)$ for $s \in \{1, 2, 3, 4\}$, and advanced asymptotic upper bounds for $s \geq 5$ utilising the Ackermann hierarchy and structural decomposition.

Contents

1 Overview and Motivation

Davenport–Schinzel sequences are combinatorial sequences that play a role in discrete geometry and the analysis of algorithms. They were originally introduced by Davenport and Schinzel (1965) [1] to model the "length" of the description of the upper envelope of a fixed number of distinct solutions to a homogeneous linear ODE of order $s \in \mathbb{N}_{\geq 1}$.

Alternatively, when $s \geq 2$, an (n, s)-Davenport–Schinzel sequence corresponds to the sequence of indices of $n \geq 0$ continuous functions (defined on a common interval $I \subset \mathbb{R}$) that constitute their lower envelope, provided that any two functions intersect at most $s - 2$ times [2]. The central question in this theory is to determine the maximum length, denoted $\lambda_s(n)$ (or `Longest_DS_func` in our formalisation), of such a sequence over any alphabet containing n distinct letters.

This project formalises the definition of these sequences, establishes their fundamental properties (including finiteness), computes exact values for small s ($s \in \{1, 2, 3, 4\}$), and sets up the machinery for general upper bounds involving the Ackermann hierarchy (specifically for $s = 5$). I follow the proofs given by Martin Klazar [3].

2 Foundations and General Theory

2.1 Definition and Motivation

I formally defined an (n, s)-Davenport–Schinzel sequence as a list u over an alphabet of size n such that:

1. No two consecutive elements are equal (sparsity).
2. It contains no alternating subsequence of length s (i.e. no subsequence of the form $a \ldots b \ldots a \ldots b \ldots$ with length s).

The associated extremal function $\lambda_s(n)$ is defined as the maximum length of such a sequence.

2.2 Finiteness

To show that $\lambda_s(n)$ is well defined, We must establish that the sequences exist and cannot be arbitrarily long. The existence is trivial (the empty sequence is an (n, s)-Davenport–Schinzel sequence for any $n \geq 0$ and $s \geq 1$). While one has the following:
Theorem (Upper Bound)**.** *For any $n \geq 0, s \geq 1$, $\lambda_s(n) \leq \left\lfloor \frac{s-1}{2} \right\rfloor n(n-1) + 1$.*

Proof Sketch (Formalised): Consider any pair of distinct symbols (a, b). How many times can they appear adjacent to each other in the sequence? If the pair appears k times, we can extract an alternating subsequence $a, b, a, b, \ldots$ of length at least $2k$. To satisfy the condition that no alternating subsequence of length s exists, we must have $2k < s$, which implies $k \leq \left\lfloor \frac{s-1}{2} \right\rfloor$. Summing this bound over all possible $\binom{n}{2}$ pairs (and accounting for order) yields the result, because the sum of the counts of all distinct pairs (a, b) appearing in the sequence can be shown to equal the length of the sequence minus 1.

2.3 Computability and Type Independence

A major part of the formalisation was making computable versions of these definitions and proving that the mathematical definition (based on sets and propositional logic) coincides with the computable one.
Theorem (Computability)**.** *The maximum length $\lambda_s(n)$ can be computed via binary search using a Boolean predicate* `is_Davenport_Schinzel_sequence_b`*.*

Theorem (Independence of Type)**.** *The value of $\lambda_s(n)$ depends only on the size of the alphabet, not on the specific type of the symbols, provided the type has at least n elements, and hence we can assume without loss of generality that the alphabet is $\mathbb{N}$.*

3 Exact Computations for Small s

I formalised the exact values of the extremal function for $s \in \{1, 2, 3, 4\}$.

3.1 Trivial Cases ($s = 1, 2, 3$)

For any $n \in \mathbb{N}$:

$$\lambda_1(n) = 0, \quad \lambda_2(n) = 1 \text{ (for } n \geq 1), \quad \lambda_3(n) = n.$$

Proof Sketch: For $s = 1$, any non-empty sequence contains an alternating subsequence of length 1. For $s = 2$, any sparse sequence of length ≥ 2 contains a, b, which is alternating of length 2. For $s = 3$, any repetition $a \dots a$ implies an alternating subsequence a, b, a of length 3; thus, elements must be unique, and so the sequence must be a chain.

3.2 The Case $s = 4$

Theorem. *For all $n \in \mathbb{N}$, $\lambda_4(n) = 2n - 1$.*

Lower Bound Construction: The sequence

$$Z_n = 1, 2, 3, \dots, n-1, n, n-1, \dots, 3, 2, 1$$

has length $2n - 1$. It is a palindrome structure increasing and then decreasing. It contains no $a \dots b \dots a \dots b$ pattern because if a appears before b in the first half, then it must appear after b in the second half; hence they cannot interleave twice.

Upper Bound Proof Sketch (Turán's Proof): I formalised the proof by Turán: We show that in any $(n, 4)$-Davenport–Schinzel sequence, there exists a symbol x appearing exactly once. Indeed, suppose that every symbol appears at least twice. Choose a symbol a appearing twice at positions $j_1 < j_2$, with j_1 maximal among all such choices. Since the sequence is sparse, we have $j_2 \neq j_1 + 1$, and thus there exists a distinct symbol b at position $j_1 + 1$. As b appears at least twice and by maximality of j_1, there must be another occurrence of b at some position $< j_1$. Hence we obtain a subsequence $b\,a\,b\,a$, which is a contradiction.

Removing one symbol x appearing once (and potentially mergeable neighbours, e.g. $\dots y, x, y \dots$ becomes $\dots y, y \dots$, in which case one of the duplicates is removed to maintain sparsity) reduces the length by at least 1 (for x) and possibly by 1 more (due to the merger). The number of distinct symbols drops at most by 2, and we obtain an $(n-1, 4)$-Davenport–Schinzel sequence. This establishes the recurrence $\lambda_4(n) \leq \lambda_4(n-1) + 2$, which implies the result because $\lambda_4(0) = 0 \leq 2 \cdot 0 - 1$ (in Lean, $a - b = 0$ if $a < b$ in `Nat`).

4 Advanced Asymptotics ($s \geq 5$)

For $s \geq 5$, the function $\lambda_s(n)$ grows slightly faster than linear, involving the inverse Ackermann function $\alpha(n)$.

4.1 The Ackermann Hierarchy

I formalised the Klazar–Ackermann hierarchy:

- $F_1(n) = 2n$
- $F_{k+1}(n) = F_k^{(n)}(1)$ (iterated function application), or equivalently $F_{k+1}(n) = F_k\left(F_{k+1}(n-1)\right)$.

The inverse Ackermann function $\alpha(n)$ is the smallest k such that $F_k(k) \geq n$.

4.2 Structural Decomposition

To prove bounds for $s = 5$, I formalised the concept of m-decomposition.
Definition (m-decomposition)**.** *A sequence u m-decomposes if it can be split into m blocks $u = u_1 \dots u_m$ such that each block u_i is a* chain *(all elements distinct) and, relative to the global* order of first appearance*, each block—after removing all elements occurring in the same position as their first occurrence in u—is decreasing.*

We defined the auxiliary function $\psi(m, n)$ as the maximum length of a sequence with at most n distinct symbols that admits an m-decomposition and avoids alternating subsequences of length 5.

4.3 The Upper Bound

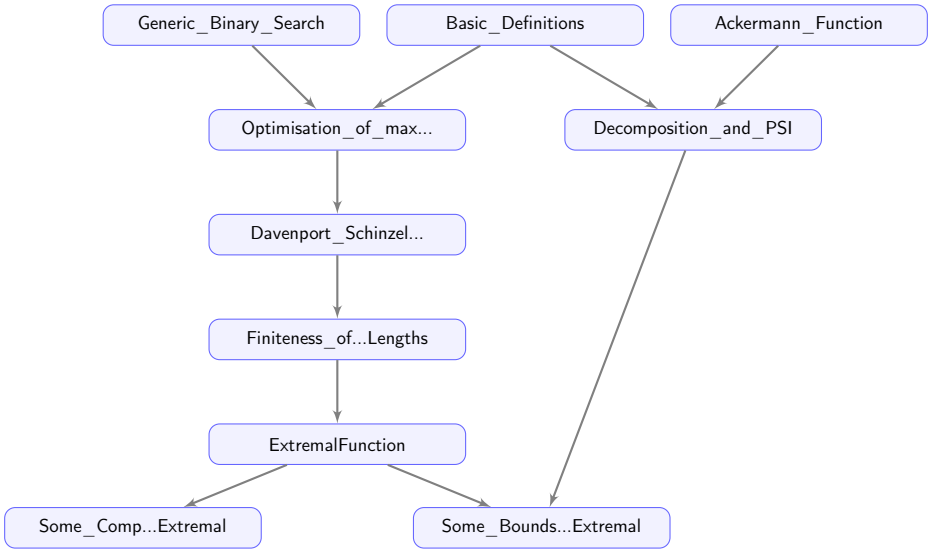
Using a recursive bound inequality for $\psi(m, n)$, We established the connection to the Ackermann function. While the definitions and the structure of the proof are formalised, some final inequalities and lemmas regarding the ψ-function and its connection to the exact Ackermann bound currently contain `sorry` in the submission.
Theorem (Bound for $s = 5$)**.**

$$\lambda_5(n) \leq 2n\alpha(n) + O\left(n\sqrt{\alpha(n)}\right).$$

5 Project Architecture and Correspondence

5.1 Dependency Graph

The following graph illustrates the exact import structure of the project (Arrows indicate dependency: A → B means B imports A).



5.2 Main Correspondences

The following table maps the main mathematical concepts to their specific implementations in the Lean 4 source files.

Mathematical Concept	**Lean Definition / Theorem**	**Source File**
	Foundations	
(n, s)-DS Sequence	`is_Davenport_Schinzel_sequence`	*Davenport...Seq.lean*
Trivial Upper Bound	`trivial_upper_bound...length`	*Finiteness...Lengths.lean*
Extremal Function $\lambda_s(n)$	`Longest_DS_func`	*ExtremalFunction.lean*
Computability of $\lambda_s(n)$	`Longest_DS_func_eq..._b`	*ExtremalFunction.lean*
Independence of Type	`Longest_DS_func_type_indep...`	*ExtremalFunction.lean*
	Exact Values	
Values for $s \in \{1, 2, 3, 4\}$	`Comp1`, ..., `Comp4`	*Some_Comp...Extremal.lean*
	Advanced Asymptotics ($s = 5$)	
Ackermann Hierarchy F_k	`KlazarAckermann.F`	*Ackermann_Function.lean*
Inverse Ackermann $\alpha(n)$	`KlazarAckermann.alpha_omega`	*Ackermann_Function.lean*
m-Decomposition	`m_decomposes`	*Decomposition...PSI.lean*
Function $\psi(m, n)$	`psi`	*Decomposition...PSI.lean*
Upper Bound for $s = 5$	`Bound5`	*Some_Bounds...Extremal.lean*

Table 1: Mapping of theorems to Lean definitions.

6 Efforts & Reflections

I organized this project across 10 files, comprising 2,406 lines of code, 940 comments, and 254 blank lines, for a total of 3,600 lines. Obviously the code is not optimized and contains some redundancy, still I believe it represents an effort on my part.

I found the connection between Constructive Computability and Mathematical Logic to be not that hard to bridge but still long. To validate examples, I needed executable code (Booleans), yet my definitions relied on propositional logic. Bridging these two worlds required me to write duplicative definitions and rigorous proofs of equivalence. In hindsight, I feel this approach was somewhat inefficient. Since Lean is not designed for high-performance computation, my algorithmic optimizations (even those yielding a $2\times$ speedup) were of limited practical use; the computation time remained significant (in my opinion). Furthermore, I primarily used computation for generating examples or checking small cases, which I arguably could have proven by hand in Mathematical Logic. While this might have been a diversion from my main goal of formalizing upper bounds, I still found establishing this correspondence intellectually satisfying. For me, it provided a concrete "bridge" between the computable and theoretical realms, reminding me of the relationship in Model Theory (e.g., Gödel's Completeness Theorem) where semantic evaluation aligns with syntactic proof. I could feel feel how the boolean `&&` (i.e., the semantic "and") corresponded to the First-Order Logic $\wedge$ (i.e., the syntactic "and"), and I still wonder how this link is actually implemented within Lean's architecture.

The most significant technical difficulty I encountered was combinatorial reasoning on lists.

Proving the boundedness of Davenport–Schinzel sequences or formalizing Turán's proof for $s = 4$ forced me to handle meticulous index manipulation often glossed over in informal proofs. I can intuitively say "remove x" in standard mathematics, but formally executing this in Lean required me to split lists (using `take` and `drop`) and rigorously prove that properties such as sparsity or the absence of forbidden patterns were preserved under these "surgeries".

I also faced a minor challenge navigating the transition between `Set` and `Finset`, though having the bridge to computable definitions helped me manage this friction.

7 Workflow and LLM Usage

I worked on this project primarily on my own, utilizing Large Language Models (specifically Gemini Pro via Copilot) as a sole assistant. Below is an overview of how I integrated these tools into my development process.

When I was unsure how to tackle a proof, or when I had a strategy that seemed excessively long to implement manually, my workflow often involved writing the proof strategy in natural language and asking the AI to help translate it into Lean or suggest appropriate tactics. Throughout this process, I also utilized search tools (primarily Loogle) and AI-assisted grep searches within VS Code to analyze existing Mathlib lemmas. For instance, I was initially unaware of the `List.IsChain` family of lemmas; discovering these via the AI significantly streamlined my construction and proofs.

However, I found that the LLMs still tended to produce unnecessarily verbose proofs. They often attempted to solve problems by reducing goals to extremely simple statements and tackling them step-by-step, frequently failing to close the goal despite the length of the code. Consequently, I often had to heavily refactor and reduce the generated proofs to make them usable.

8 Conclusion and Future Work

The formalisation appears (to me) to be structurally correct, though it currently relies on a few remaining (difficult) `sorry`s to be fully complete. I am aware that the code is currently "unidiomatic". Future work may probably involve refactoring the proofs—and perhaps rethinking the architectural design—to clean up the code. This refactoring is essential if I hope to eventually submit this work to Mathlib or CSLib.

References

[1] Harold Davenport and Andrzej Schinzel. A combinatorial problem connected with differential equations. *American Journal of Mathematics*, 87(3):684–694, 1965. URL https://www.jstor.org/stable/2373068.

[2] ETH Zurich. Chapter 14: Davenport-schinzel sequences. Lecture Notes for Computational Geometry (CG 2012), 2012. URL https://ti.inf.ethz.ch/ew/lehre/CG12/lecture/Chapter%2014.pdf. Department of Computer Science, ETH Zurich.

[3] Martin Klazar. On the maximum lengths of davenport-schinzel sequences. In Ronald L. Graham, Jan Kratochvíl, Jaroslav Nešetřil, and Fred S. Roberts, editors, *Contemporary Trends in Discrete Mathematics*, volume 49 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 169–178. American Mathematical Society, Providence, RI, 1999. URL https://kam.mff.cuni.cz/~klazar/maxleng.pdf.