Class 1 - Project Overview

Guiding Genetic Algorithms with Language Models for Combinatorial Optimization

MISE Research Program - July 2025

Pedro Paredes

Who am I?

Call me Pedro

Pedro Paredes

Teaching Professor at Princeton University

PhD in Computer Science from CMU

Been teaching for MISE for 5 years

Class logistics

Website https://www.cs.princeton.edu/~pparedes/teaching/mise/summer25/

Email pparedes@cs.princeton.edu

Please ask questions during class! Raise your arm through zoom or unmute yourselves and talk!

Why this project?

Problems as Big Puzzles

A ***combinatorial problem*** can be viewed as an exceptionally large puzzle:

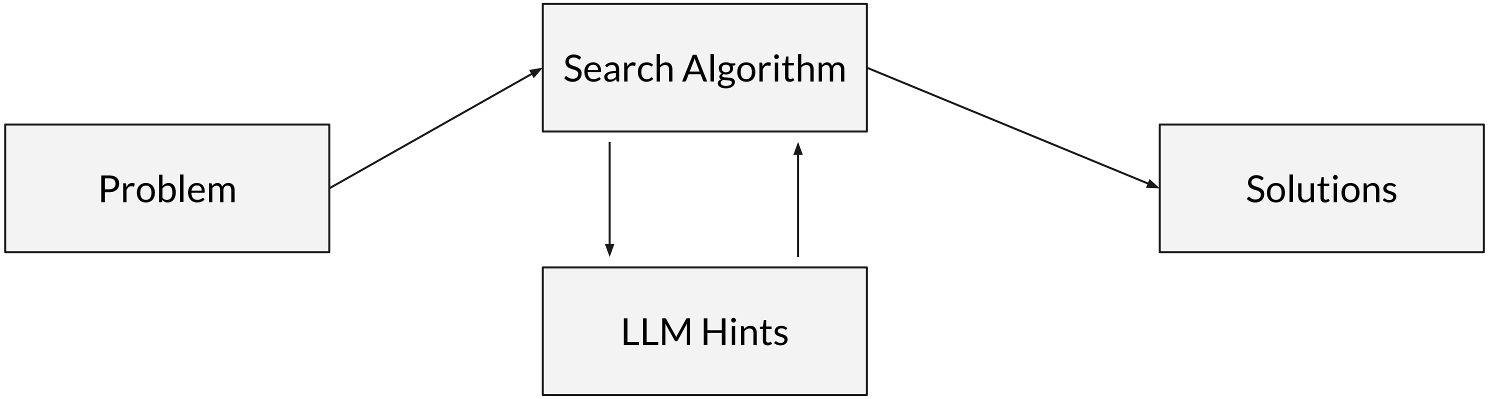
- We are given many discrete pieces and clear rules for how they may be arranged
- Our goal is to discover a valid arrangement that is best according to some measure of quality

Combinatorial problems appear everywhere: packing delivery trucks, arranging playlists, balancing sports teams, or routing thousands of packages through a warehouse, …

What Are We Building?

We will create an algorithm that ***searches*** through arrangements of pieces to solve these “puzzles”

Our algorithm will use large language models to get advice on where to search next



Combinatorial Optimization

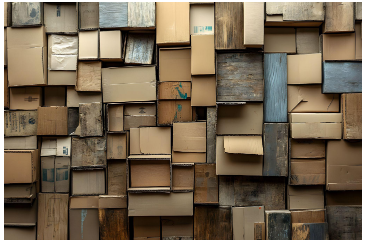
A Big Puzzle - Packing a Truck

Problem: A courier arrives with a pile of packages that must all go into a single delivery truck. Some are large, some small, some oddly shaped. Load everything so nothing gets crushed and no space is wasted.

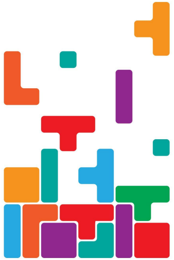
Slightly technical framing:

- *Puzzle pieces:* a fixed set of packages, each with known dimensions.
- *Clear rules (constraints):* packages may not overlap and must fit in the truck (W × H × D size box)
- *Measure of quality (objective):* maximize the total packed volume.

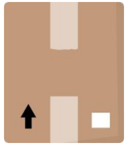
Finding the optimal loading order and orientation for all packages - while obeying the constraints - is a classic combinatorial optimization problem.



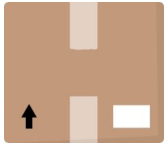
Just like a big game of tetris!



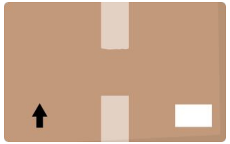
A Big Puzzle - Packing a Truck



x 9



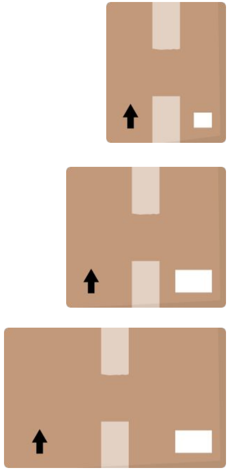
x 5



x 3

Fit as many as possible here:

A Big Puzzle - Packing a Truck

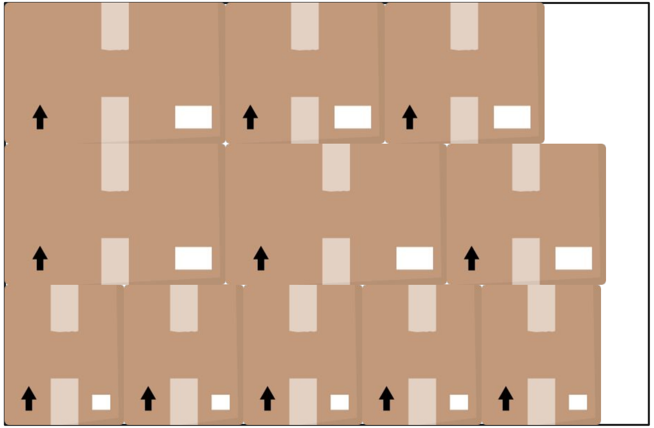


x 9

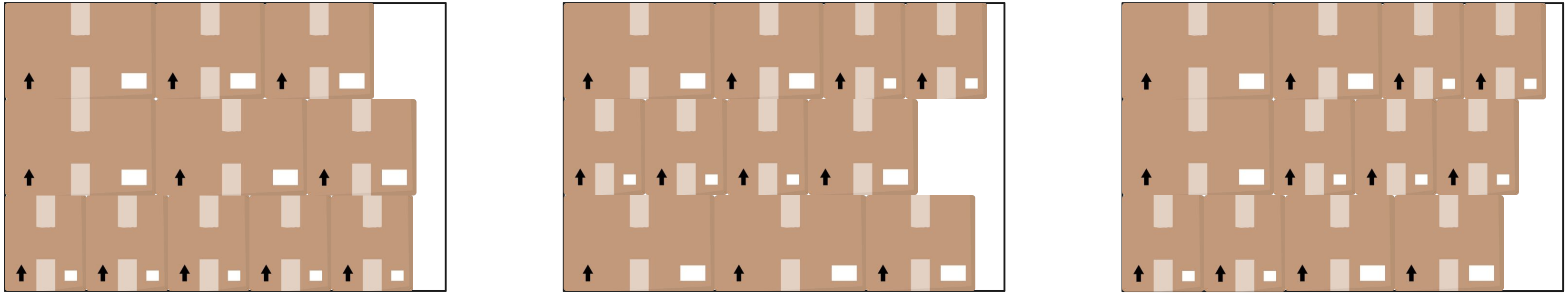
x 5

x 3

Possible arrangement with: 3 large, 3 medium, 5 small



A Big Puzzle - Packing a Truck



More than 1 million possible arrangements - computer finds best in ~10 seconds

Searching For Solutions

Search Algorithms

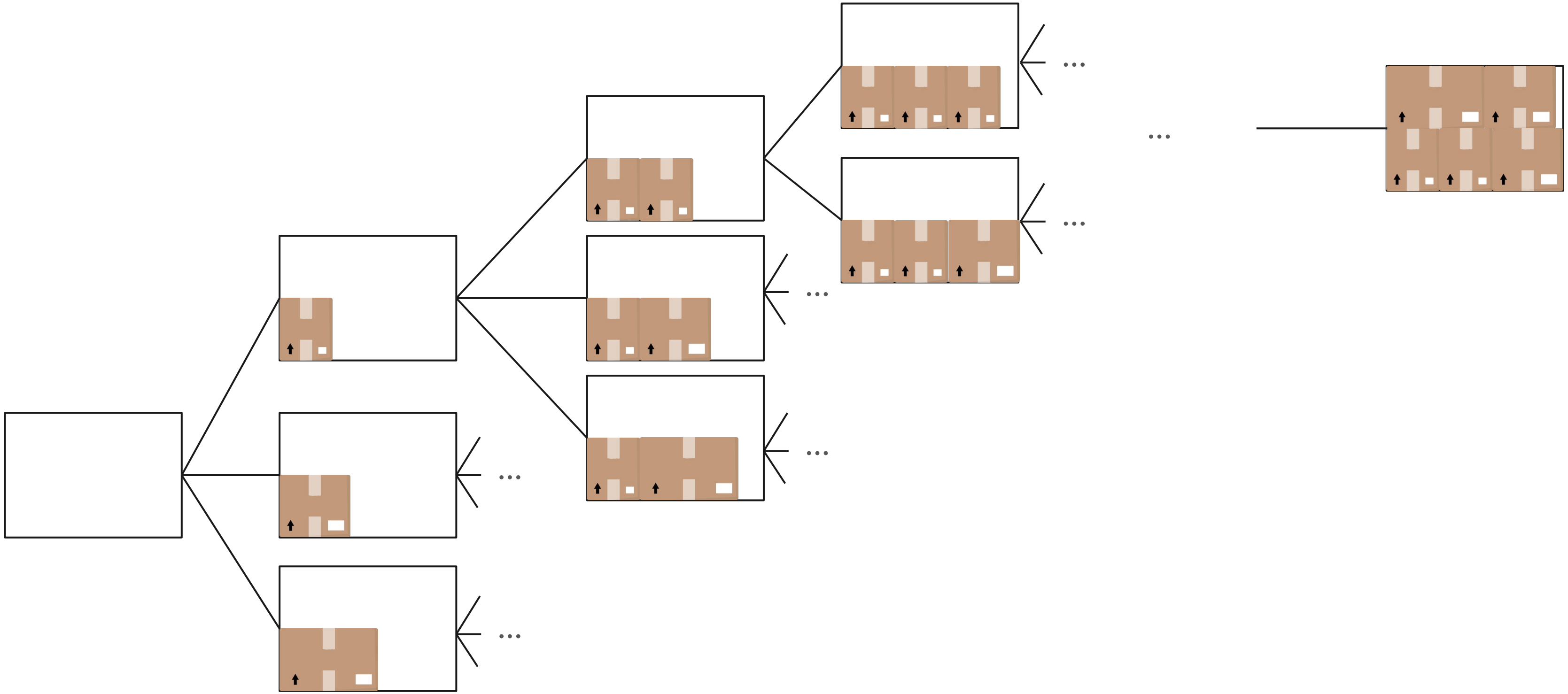
A ***search algorithm*** is a systematic way to explore the many possible arrangements:

- Generate a candidate arrangement
- Evaluate it with the objective function (score)
- Decide what to try next, based on a strategy - repeat until found best arrangement

The most straightforward strategy is ***exhaustive search***:

- Enumerate every valid arrangement, evaluate each, and keep the best

Search Algorithms - Exhaustive Search



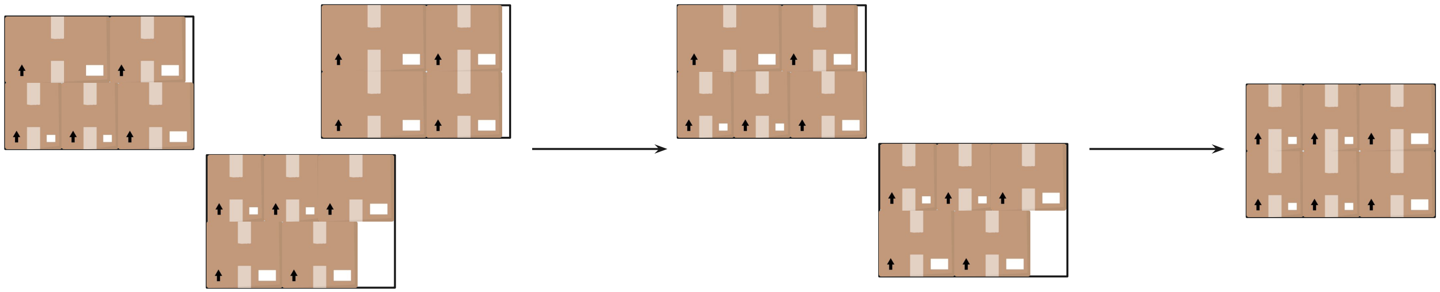
Search Algorithms - Heuristics

We make search more efficient using ***heuristics***: rules of thumb that steer the search toward good solutions. Examples:

- Check more promising solutions first
- Discard solutions that we can be sure won't be the best

We will use a method known as a **Genetic Algorithm:**

- Keep a population of candidate solutions
- Evaluate fitness (i.e. who's more promising?) and select the stronger candidates
- Combine parts of two solutions (crossover) and mutate them randomly



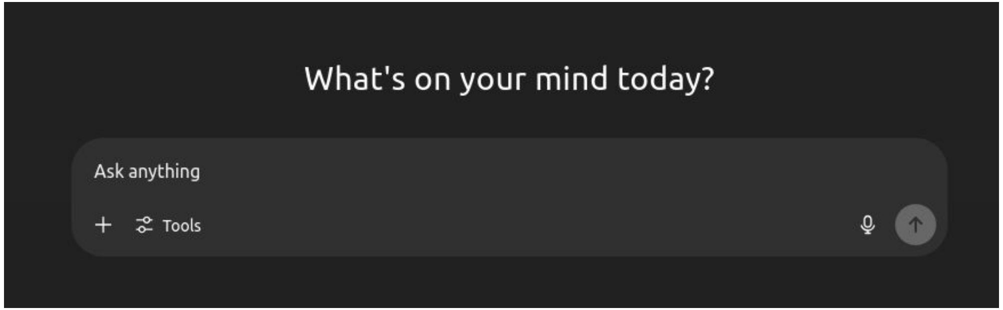
Using Language Models

Large Language Models

A large language model (LLM) is an AI model trained on vast collections of text and code

Capabilities we care about:

- Text generation — produces coherent explanations or documentation.
- Code synthesis — writes or edits functions on demand.
- Pattern completion — given partial input, can propose plausible continuations (useful for repairing or mutating programs).



LLM-Guided Evolution

Large language models can propose smarter mutations and even entire heuristic functions

This project is inspired by the following two works, by the Google DeepMind team

You will essentially build a simpler version of their method and apply it to some new problems

Article

Mathematical discoveries from program search with large language models

https://doi.org/10.1038/s41586-023-06924-6
Received: 12 August 2023
Accepted: 30 November 2023
Published online: 14 December 2023
Open access
Check for updates

Bernardino Romera-Paredes[1,4], Mohammadamin Barekatain[1,4], Alexander Novikov[1,4], Matej Balog[1,4], M. Pawan Kumar[1,4], Emilien Dupont[1,4], Francisco J. R. Ruiz[1,4], Jordan S. Ellenberg[2], Pengming Wang[1], Omar Fawzi[3], Pushmeet Kohli[1] & Alhussein Fawzi[1,4]

Large language models (LLMs) have demonstrated tremendous capabilities in solving complex tasks, from quantitative reasoning to understanding natural language. However, LLMs sometimes suffer from confabulations (or hallucinations), which can result in them making plausible but incorrect statements[1,2]. This hinders the use of current large models in scientific discovery. Here we introduce FunSearch (short for searching in the function space), an evolutionary procedure based on pairing a pretrained LLM with a systematic evaluator. We demonstrate the effectiveness of this approach to surpass the best-known results in important problems, pushing the boundary of existing LLM-based approaches[3]. Applying FunSearch to a central problem in extremal combinatorics—the cap set problem—we discover new constructions of large cap sets going beyond the best-known ones, both in finite dimensional and asymptotic cases. This shows that it is possible to make discoveries for established open problems using LLMs. We showcase the generality of FunSearch by applying it to an algorithmic problem, online bin packing, finding new heuristics that improve on widely used baselines. In contrast to most computer search approaches, FunSearch searches for programs that describe how to solve a problem, rather than what the solution is. Beyond being an effective and scalable strategy, discovered programs tend to be more interpretable than raw solutions, enabling feedback loops between domain experts and FunSearch, and the deployment of such programs in real-world applications.

Many problems in mathematical sciences are 'easy to evaluate', despite being typically 'hard to solve'. For example, in computer science, NP-complete optimization problems admit a polynomial-time evaluation procedure (measuring the quality of the solution), despite the widespread belief that no polynomial-time algorithms to solve such problems exist. We focus in this paper on problems admitting an efficient 'evaluate' function, which measures the quality of a candidate solution. Prominent examples include the maximum independent set problem and maximum constraint satisfaction problems (such as finding the ground state energy of a Hamiltonian). Our goal is to generate a 'solve' program, such that its outputs receive high scores from the 'evaluate' function (when executed on inputs of interest), and ultimately improve on the best-known solutions.

Whereas large language models (LLMs) have recently seen notable improvements in their coding capabilities[4–8], with applications including debugging[9,10], solving code competitions[11,12] and improving code performance[13], synthesizing 'solve' programs for open problems requires finding new ideas that are verifiably correct. This is very hard for LLMs, as they tend to confabulate or ultimately fall short of going beyond existing results. To surpass the 'nominal' capabilities of LLMs, recent studies[3] have combined them with evolutionary algorithms[14,15], leading to important improvements on diverse synthetic problems[16], searching for neural network architectures[17–19] and solving puzzles[20]. Our proposed method, FunSearch, pushes the boundary of LLM-guided evolutionary procedures to a new level: the discovery of new scientific results for established open problems and the discovery of new algorithms. Surpassing state-of-the-art results on established open problems provides a clear indication that the discoveries are truly new, as opposed to being retrieved from the LLM's training data.

FunSearch (short for searching in the function space) combines a pretrained (frozen) LLM, whose goal is to provide creative solutions, with an evaluator, which guards against confabulations and incorrect ideas. FunSearch iterates over these two components, evolving initial low-scoring programs into high-scoring ones discovering new knowledge. Key to the success of this simple procedure is a combination of several essential ingredients. First, we sample best performing programs and feed them back into prompts for the LLM to improve on; we refer to this as best-shot prompting. Second, we start with a program in the form of a skeleton (containing boilerplate code and potentially known structure about the problem), and only evolve the part governing the critical program logic. For example, by setting a greedy program skeleton, we evolve a priority function used to make decisions at every

[1]Google DeepMind, London, UK. [2]Department of Mathematics, University of Wisconsin-Madison, Madison, WI, USA. [3]Laboratoire de l'Informatique du Parallélisme, University of Lyon (Inria, ENS Lyon, UCBL, LIP), Lyon, France. [4]These authors contributed equally: Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Alhussein Fawzi. e-mail: brp@google.com; pushmeet@google.com; afawzi@google.com

468 | Nature | Vol 625 | 18 January 2024

Google DeepMind

***AlphaEvolve*: A coding agent for scientific and algorithmic discovery**

Alexander Novikov*, Ngân Vũ*, Marvin Eisenberger*, Emilien Dupont*, Po-Sen Huang*, Adam Zsolt Wagner*, Sergey Shirobokov*, Borislav Kozlovskii*, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli and Matej Balog*
Google DeepMind[1]

In this white paper, we present *AlphaEvolve*, an evolutionary coding agent that substantially enhances capabilities of state-of-the-art LLMs on highly challenging tasks such as tackling open scientific problems or optimizing critical pieces of computational infrastructure. *AlphaEvolve* orchestrates an autonomous pipeline of LLMs, whose task is to improve an algorithm by making direct changes to the code. Using an evolutionary approach, continuously receiving feedback from one or more evaluators, *AlphaEvolve* iteratively improves the algorithm, potentially leading to new scientific and practical discoveries. We demonstrate the broad applicability of this approach by applying it to a number of important computational problems. When applied to optimizing critical components of large-scale computational stacks at Google, *AlphaEvolve* developed a more efficient scheduling algorithm for data centers, found a functionally equivalent simplification in the circuit design of hardware accelerators, and accelerated the training of the LLM underpinning *AlphaEvolve* itself. Furthermore, *AlphaEvolve* discovered novel, provably correct algorithms that surpass state-of-the-art solutions on a spectrum of problems in mathematics and computer science, significantly expanding the scope of prior automated discovery methods (Romera-Paredes et al., 2023). Notably, *AlphaEvolve* developed a search algorithm that found a procedure to multiply two 4×4 complex-valued matrices using 48 scalar multiplications; offering the first improvement, after 56 years, over Strassen's algorithm in this setting. We believe *AlphaEvolve* and coding agents like it can have a significant impact in improving solutions of problems across many areas of science and computation.

1. Introduction

Discovering new high-value knowledge, such as making a novel scientific discovery or developing a commercially valuable algorithm, generally requires a prolonged process of ideation, exploration, backtracking on unpromising hypotheses, experimentation, and validation. There has been much recent interest in using large language models (LLMs) to automate significant parts of this process. Hopes of success here are driven by the breathtaking power of recent LLMs [32, 76], which can enhance their capabilities using test-time compute, and the rise of *agents* that combine language generation and action [88, 114]. These advances have improved performance across a range of established benchmarks and accelerated discovery-oriented tasks like hypothesis generation [34] and experiment design [7, 43]. However, getting LLM pipelines all the way to making entirely new scientific or practical discoveries remains challenging.

In this white paper, we present an LLM code superoptimization agent, called *AlphaEvolve*, that takes on this challenge using a combination of evolutionary computation and LLM-based code generation. *AlphaEvolve* focuses on the broad spectrum of scientific and engineering

[1]See Acknowledgments and Author information section. *Equal contributions.

© 2025 Google DeepMind. All rights reserved

Romera-Paredes, Bernardino, et al. "Mathematical discoveries from program search with large language models." Nature 625.7995 (2024): 468-475.

Novikov, Alexander, et al. "AlphaEvolve: A coding agent for scientific and algorithmic discovery."

Roadmap

Course Roadmap

Date	Topic
July 14	Project Overview
July 16	Python Programming Review I
July 18	Python Programming Review II
July 19	Greedy Heuristics and Hill Climbing
July 21	Introduction to Genetic Algorithms
July 22	Implementing a Genetic Algorithm in Python
July 24	Analyzing and Plotting Results
July 25	Problem Focus: Conveyor Diverter Routing
July 28	Combining LLMs and GAs
July 29	Exploration & Comparison: GA vs GA+LLM
July 30	Report Writing & Analysis
July 31	Final Presentation

Note: schedule might change slightly. I'll let you know a few days in advance if so

Warmup Tasks

Basic Stats

Write a function `stats(nums)` that returns a tuple `(total, average)` for a list of integers.

```python
def stats(nums):
    total = 0
    count = 0

    for n in nums:
        total += n
        count += 1

    if count == 0:
        return (0, 0)

    average = total / count
    return (total, average)
```

Second-Largest Distinct Value

Implement `second_largest(nums)` that returns the second-largest **distinct** integer in the list. Assume that all elements are positive.

```python
def second_largest(nums):
    largest = 0
    second = 0

    for n in nums:
        if n == largest:
            continue

        if largest == 0 or n > largest:
            second = largest
            largest = n

        elif second == 0 or n > second:
            second = n

    return second
```

Right Rotation by k

Given a list nums and non-negative integer k, return a **new list** that is nums rotated to the right by k positions (e.g., [1,2,3,4], k=1 → [4,1,2,3]).

```python
def rotate_right(nums, k):
    n = len(nums)
    if n == 0:
        return []
    k %= n
    res = []
    for i in range(n - k, n):
        res.append(nums[i])
    for i in range(n - k):
        res.append(nums[i])
    return res
```

Minimum Absolute Difference Pair

Write `min_diff(nums)` that returns the **smallest absolute difference** between any two distinct numbers in `nums`.

```python
def min_diff_sort(nums):
    n = len(nums)
    if n < 2:
        return None
    best = abs(nums[1] - nums[0])
    for i in range(n):
        for j in range(i + 1, n):
            d = abs(nums[i] - nums[j])
            if d < best:
                best = d
    return best
```

```python
def min_diff_sort_fast(nums):
    if len(nums) < 2:
        return None
    nums = sorted(nums)
    best = abs(nums[1] - nums[0])
    for i in range(2, len(nums)):
        d = abs(nums[i] - nums[i - 1])
        if d < best:
            best = d
    return best
```

Extra Tasks

Solve by Next Class

1. Letter-Frequency Histogram

Write `letter_histogram(text)` that prints each **alphabetic** character (case-insensitive) and its count in `text`. Ignore digits, punctuation, and spaces. Print the histogram in **descending count order**, and for ties, alphabetically.

2. Run-Length Encoder / Decoder

Implement two functions: `compress(s)` returns the run-length encoding of string `s` (e.g., `"aaabbc"` → `"a3b2c1"`). `decompress(rle)` reverses the process. Assume runs never exceed 9.

3. Print Matrix in Spiral Order

Implement `spiral_print(n)` that returns a rectangular 2-D list with numbers `1` through `n*n` in clockwise spiral order. Example: `3 -> [[1,2,3],[8,9,4],[7,6,5]]`.

Solve by Next Class

4. Tic-Tac-Toe Winner

Function `winner(board)` gets a 3 × 3 list of lists containing `'X'`, `'O'`, or `''`. Return `'X'`, `'O'`, or `None` depending on who has three in a row (rows, columns, or diagonals). No need to check invalid games.

5. Merge Overlapping Intervals

Given a list of half-open intervals `[(start, end), ...]` with `start < end`, write `merge(intervals)` that returns a new list where overlapping or touching intervals are combined. Sort the final list by start time.

6. Balanced Brackets

Implement `is_balanced(expr)` that returns `True` if every `'('` has a matching `')'` in the proper order, otherwise `False`. Assume that `expr` is a string containing only parenthesis.

What's next?

Next class on Wednesday (no class tomorrow)

Write Python code to solve each of the extra tasks (we will discuss solutions next class)

Class 2: Python Programming Review I

Image References

- Warehouse - Adobe Stock - Education License
- Soccer - Adobe Stock - Education License
- Van - Adobe Stock - Education License
- Music - Adobe Stock - Education License
- Boxes - Adobe Stock - Education License
- Tetris - Adobe Stock - Education License