Task: ACT
Accelerated Command Training

AACPP WiSe 2025/26 Round 6 Memory: 2GiB 2026.01.14 – 2026.01.28

Dexter is a very smart cat and knows a whole bunch of tricks. Mat sometimes grabs a pack of snacks and gives him various commands. A successful trick always earns a snack, so Dexter is very eager to follow all the commands.

However, Dexter is also a very ~~lazy~~ efficient cat. Does he really need to learn all the commands exactly? He would much rather look for some simple, short patterns in the complex commands and learn those instead.

Dexter has a list of all the commands that Mat can give him. He needs your help: he will give you a pattern and ask **in how many different commands the pattern appears as a substring**.

Communication

This is an interactive task. Your program *must* use a library to communicate with Dexter. Your program *must not* read from standard input or write anything to standard output.

- In **Rust**, declare the library as a module with `mod actlib` at the top of the file and use the provided `actlib::ActLib` struct.
- In **C++**, include the provided `actlib.hpp` header and use the `ActLib` class.

The library provides six functions.

- `new()`
  create a new instance of `ActLib`. You should create exactly one instance at the start of the program and use it throughout execution.
  - in **Rust**: the `ActLib::new` function.
  - in **C++**: the default `ActLib` constructor;
- `get_n()`
  returns the parameter n – the total number of commands to consider.
  - in **Rust**: the `ActLib::get_n(&self) -> usize` method.
  - in **C++**: the `size_t ActLib::get_n() const` method;
- `get_m()`
  returns the parameter m – the total number of patterns to consider.
  - in **C++**: the `size_t ActLib::get_m() const` method;
  - in **Rust**: the `ActLib::get_m(&self) -> usize` method.
- `get_command(i)`
  returns the i-th command from the input, zero-indexed. The given i parameter must be between 0 and $n-1$. All commands consist of 1 or more lowercase ASCII letters (a to z).
  - in **Rust**: the `ActLib::get_command(&self, i: usize) -> &[u8]` method.
  - in **C++**: the `std::string_view ActLib::get_command(size_t i) const` method;
- `get_current_pattern()`
  returns the pattern for which Dexter wants an answer. All patterns consist of 1 or more lowercase ASCII letters (a to z). There are m patterns. At the start Dexter asks for pattern number 0. Repeated calls to this function will keep returning the same pattern 0 until `answer(x)` is called, at which point Dexter will advance to the next pattern. After m answers this function *must not* be called anymore.
  - in **Rust**: the `ActLib::get_current_pattern(&self) -> &[u8]` method.
  - in **C++**: the `std::string_view ActLib::get_current_pattern() const` method;

- `answer(x)`
  answers Dexter that the pattern being currently considered matches x commands. This function must be called exactly m times, where i-th call constitutes the answer to the i-th pattern.
  - in **Rust**: the `ActLib::answer(&mut self, result: usize)` method.
  - in **C++**: the `void ActLib::answer(size_t result)` method;

An example library implementation is available in the Downloads section as `actlib.cpp` and `actlib.rs`. It reads test inputs from standard input and writes all answers to the standard output. To compile your program with the library:

- in **Rust**: you can use the same command as normal, `rustc` will automatically discover the `actlib.rs` file in the same directory;
- in **C++**: add `actlib.cpp` to the compilation command:
  ```
  g++ -std=c++23 -static -O2 -s -lm -o a.out a.cpp actlib.cpp
  ```

You can also find example (incorrect) solutions that nonetheless correctly use the library: `actex.cpp` and `actex.rs`.

The example library is not the same as used during grading. You cannot rely on its implementation details, only on the public API described in this section.

Example

Here we present an example program execution for the test `act0.in`, showing which library functions are called from the main program and its return values.

Function call	Return value	Description
`new()`	`ActLib` object	Initialisation of the library.
`get_n()`	2	There are 2 commands to consider.
`get_command(0)`	$give$	First command is “give”.
`get_command(1)`	$leave$	First command is “leave”.
`get_m()`	3	There are 3 patterns to answer to.
`get_current_pattern()`	eav	First pattern is “eav”.
`answer(1)`	—	Only the second command matches the pattern.
`get_current_pattern()`	ve	Second pattern is “ve”.
`answer(2)`	—	Both commands match the pattern.
`get_current_pattern()`	eve	Third and last pattern is “eve”.
`answer(0)`	—	None of the commands match the pattern.

All three patterns received an answer and the program may now terminate.

Additional examples

The following initial tests are also available:

- `0b` – small sample, $n = 5$, $m = 3$.
- `0c` – $n = 10$, $m = 10$, every command has length 100 and is a single repeating letter from a to j. The i-th pattern is exactly the i-th command.
- `0d` – $n = 100$, $m = 65\,536$, the i-th command is $a^{100-i}b^i$, the patterns are all possible strings of length 16 over $\{a, b\}$.
- `0e` – $n = 1\,000$, $m = 100\,000$, the i-th command is a^{i+1}, all patterns are the same string a^{44}.

Limits

Your solution will be evaluated on a number of hidden test cases divided into groups. Points for a group are awarded if and only if the submission returns the correct answer for each of the tests in the group within the allotted time limit. These groups are organised into subtasks with the following limits and points awarded.

In the following table S is the sum of lengths of all strings in the input (commands and patterns). In every test $n, m \geq 1$.

Subtask	Limits	Points
1.	$n \leq 100$, $m \leq 100$, $S \leq 5\,000$	2
2.	$n \leq 100$, $m \leq 100\,000$, $S \leq 1\,000\,000$	2
3.	$n \leq 1\,000$, $m \leq 100\,000$, $S \leq 5\,000\,000$	6