

Substituting this back into the expression for  $(S_n)$ , we get:

$$S_n = \sum_{k=1}^n \sqrt{(2k-1)^2 + \left(\frac{17(2k-1)}{n^2}\right)^2} = \sqrt{n^4 + 289}$$

We need  $(\sqrt{n^4 + 289})$  to be an integer. Let  $(m)$  be this integer, so:

$$m^2 = n^4 + 289$$

$$m^2 - n^4 = 289$$

This can be factored as:

$$(m - n^2)(m + n^2) = 289$$

The factor pairs of 289 are (1, 289) and (17, 17). The valid factor pair is (1, 289), leading to:

$$m - n^2 = 1$$

$$m + n^2 = 289$$

Adding and subtracting these equations, we find:

$$2m = 290 \implies m = 145$$

$$2n^2 = 288 \implies n^2 = 144 \implies n = 12$$

Thus, the unique positive integer  $(n)$  is  $\boxed{12}$ .

#### Listing 6 | An example SFT trajectory from reasoning data related to code.

```
## Question
In this kata, your job is to create a class Dictionary which you can add words to and
their entries. Example:
'''python
>>> d = Dictionary()

>>> d.newentry('Apple', 'A fruit that grows on trees')

>>> print(d.look('Apple'))
A fruit that grows on trees

>>> print(d.look('Banana'))
Can't find entry for Banana
'''

Good luck and happy coding!

Here is the setup code:
'''python
class Dictionary():
    def __init__(self):
        # Your code
        pass

    def newentry(self, word, definition):
        # Your code
        pass

    def look(self, key):
        # your code
        pass
'''

## Response
<think>
Okay, I need to create a Python class called Dictionary. The class should allow adding
```

words with their definitions and looking them up. Hmm, let's think about the setup.

So, when a Dictionary object is created, like `d = Dictionary()`, I probably need some kind of data structure to store the words and their entries. A dictionary seems perfect for this because it can hold key-value pairs, where the key is the word and the value is the definition.

So in the `__init__` method, I should initialize an empty dictionary. Like `self.entries = {}`.

Then, there's the `newentry` method. This method takes two arguments: word and definition. So in `newentry`, I need to add the word as the key and the definition as the value to the `self.entries` dictionary.

Now, the `look` method is called with a key. It needs to return the definition if the key exists, else return a string saying it can't find the entry. Like, for example, looking up 'Apple' returns the definition, but looking up 'Banana' returns 'Can't find entry for Banana'.

Wait, how does it handle case sensitivity? I guess we're assuming the keys are case-sensitive as per the string passed. So 'Apple' is different from 'apple'.

So the steps for the `look` function are: check if the key is in `self.entries`. If yes, return the value. If no, return the specific string.

Putting it all together:

For the `__init__`:

```
def __init__(self):  
    self.entries = {}
```

For `newentry`:

```
def newentry(self, word, definition):  
    self.entries[word] = definition
```

For `look`:

```
def look(self, key):  
    if key in self.entries:  
        return self.entries[key]  
    else:  
        return f"Can't find entry for {key}"
```

Wait, but in the example, when `print(d.look('Banana'))`, it prints the message without quotes. So the method returns the string which when printed gives that output.

So, yeah, that's correct.

Let me test this logic.

Testing with the example:

```
d = Dictionary()  
d.newentry('Apple', 'A fruit...') --> adds to entries.  
d.look('Apple') returns the string, so print shows it correctly.  
d.look('Banana') returns the can't find message.
```

That should work.

I think that's the basic plan. So the setup code is given, I have to fill in the methods. So in code, it's as I wrote.

Wait, any edge cases? Like, what if word is an empty string? Or any other type? The problem doesn't specify handling those, so I think the code should work as intended.

Another thing: in the look method, when we do 'if key in self.entries', does it check for the presence as a key? Yes, so that's correct.

So, to implement:

Start by adding the `__init__`, `newentry`, and `look` methods as described.  
</think>

To solve this problem, we need to create a Python class called `Dictionary` that allows adding words with their definitions and looking them up. The class should return the definition if the word exists and a specific message if it does not.

### Approach

1. **\*\*Data Structure\*\***: Use a dictionary to store the words as keys and their definitions as values. This allows efficient lookups and additions.
2. **\*\*Initialization\*\***: In the `'__init__'` method, initialize an empty dictionary to store the word definitions.
3. **\*\*Adding Entries\*\***: The `'newentry'` method takes a word and its definition, then adds this pair to the dictionary.
4. **\*\*Looking Up Entries\*\***: The `'look'` method checks if the word exists in the dictionary. If it does, it returns the definition; otherwise, it returns a message indicating the word was not found.

### Solution Code

```
'''python
class Dictionary():
    def __init__(self):
        self.entries = {}

    def newentry(self, word, definition):
        self.entries[word] = definition

    def look(self, key):
        if key in self.entries:
            return self.entries[key]
        else:
            return f"Can't find entry for {key}"
'''
```

### Explanation

1. **\*\*Initialization\*\***: The `'__init__'` method sets up an empty dictionary `'self.entries'` to store the words and their definitions.
2. **\*\*Adding Entries\*\***: The `'newentry'` method takes two parameters, `'word'` and `'definition'`, and adds them to `'self.entries'` where `'word'` is the key and `'definition'` is the value.
3. **\*\*Looking Up Entries\*\***: The `'look'` method checks if the given key exists in `'self.entries'`. If the key is found, its corresponding definition is returned. If not, a message is returned indicating the absence of the entry.

This approach ensures efficient storage and retrieval of word definitions using a dictionary data structure, providing an optimal solution for the problem.

---

Listing 7 | An example SFT trajectory from non-reasoning data related to writing.

## Question

write a short poem ending in "was tentacles all along"

```

## Response
<think>
Okay, I need to help the user write a short poem that ends with "was tentacles all
along." Hmm, ending with that specific line suggests a twist or surprise, maybe
something unexpected.

First, I'll think about the theme. Tentacles make me think of sea creatures, like
octopuses or squids, so perhaps an underwater setting would work well. Maybe a
treasure hunt or an adventure beneath the waves could set the scene.

I want the poem to build up anticipation. Starting with something mysterious, like
shadows under the sea, could create intrigue. Then, introducing elements like a
sunken chest or sailors' lore might add depth and keep the reader engaged.

Rhyme scheme is important for a poem. I'll probably go with something simple, like AABB
or ABAB, to keep it flowing smoothly. Each stanza should have four lines to
maintain structure.

Now, the twist at the end should be sudden but make sense. Perhaps the treasure they
seek turns out to be something entirely different. The last line needs to be
impactful, so building up the suspense through the previous lines will make the
ending more surprising.

Let me outline the stanzas. The first stanza could set the scene with the ocean depths
and a hidden secret. The second stanza can show the action, like diving for
treasure, and then reveal the unexpected tentacles.

I'll make sure each line flows naturally and the imagery is vivid. Using words like "
waves," "shadows," "caverns," and "mystery" can help paint the picture. Ending with
the twist will leave the reader with a lasting impression.

Alright, putting it all together, I'll craft two stanzas that build up to the final
line, ensuring the surprise feels satisfying and the poem feels complete.
</think>

Beneath the waves, where shadows creep,
A secret stirs in the ocean deep.
A sunken chest, a sailor's song,
The mystery whispered all along.

We dove for pearls, for gold, for lore,
Through caverns dark, to the ocean floor.
But what we found, so strange, so strong--
It wasn't treasure... it was tentacles all along.

```

## B.4. Hyper-Parameters

### B.4.1. Hyper-Parameters of DeepSeek-R1-Zero-Qwen-32B

To train DeepSeek-R1-Zero-Qwen-32B, we set the learning rate to  $2e-6$ , the KL coefficient to 0.001, and the sampling temperature to 1 for rollout. For each question, we sample 16 outputs with a maximum length of 32,768. Each training step consists of 32 unique questions, resulting in a training batch size of 512 per step. Every 400 steps, we replace the reference model with the latest policy model. To accelerate training, each rollout generates 8,192 outputs, which are randomly split into 16 mini-batches and trained for only a single inner epoch.

#### B.4.2. Hyper-Parameters of SFT

For code-start SFT and the second-stage SFT, we fine-tune DeepSeek-V3-Base for 2-3 epochs using the curated dataset, as described in B.3. We employ a cosine decay learning rate scheduler, starting at  $5 \times 10^{-5}$  and gradually decreasing to  $5 \times 10^{-6}$ . The maximum context length is set to 32,768 tokens, and the batch size is 128.

#### B.4.3. Hyper-Parameters of Distillation

For distillation, we fine-tune the corresponding base model for 2–3 epochs using the 800k data described in Section B.3.3. The base model and initial learning rate are listed in Table 6. We employ a cosine decay learning rate scheduler that gradually decreases the learning rate to one-tenth of its initial value. The maximum context length is 32,768 tokens, and the batch size is 64.

Table 6 | DeepSeek-R1 Distilled Models, their corresponding Base Models, and Initial Learning Rates.

Distilled Model	Base Model	Initial Learning Rate
DeepSeek-R1-Distill-Qwen-1.5B	Qwen2.5-Math-1.5B	$1 \times 10^{-4}$
DeepSeek-R1-Distill-Qwen-7B	Qwen2.5-Math-7B	$8 \times 10^{-5}$
DeepSeek-R1-Distill-Qwen-14B	Qwen2.5-14B	$7 \times 10^{-5}$
DeepSeek-R1-Distill-Qwen-32B	Qwen2.5-32B	$6 \times 10^{-5}$
DeepSeek-R1-Distill-Llama-8B	Llama-3.1-8B	$5 \times 10^{-5}$
DeepSeek-R1-Distill-Llama-70B	Llama-3.3-70B-Instruct	$2 \times 10^{-5}$

#### B.4.4. Training Cost

Regarding our research on DeepSeek-R1, we utilized the A100 GPUs to prepare for the experiments with a smaller model (30B parameters). The results from this smaller model have been promising, which has allowed us to confidently scale up to 660B R1-Zero and R1. For the training of DeepSeek-R1-Zero, we employed 64\*8 H800 GPUs, and the process required approximately 198 hours. Additionally, during the training phase of DeepSeek-R1, we utilized the same 64\*8 H800 GPUs, completing the process in about 4 days, or roughly 80 hours. To create the SFT datasets, we use 5K GPU hours. The details are shown in Table 7.

### B.5. Reward Hacking

In the context of LLM training, reward hacking refers to the phenomenon wherein a model exploits flaws or biases in the reward function, thereby achieving high reward scores without truly aligning with the underlying human intent. In our work, we observe such reward hacking behavior when employing the helpful reward model. Specifically, if the reward model contains systematic biases or inaccuracies, the LLM may learn to generate responses that are rated highly by the model but diverge from authentic human preferences. This misalignment can manifest in performance degradation on tasks requiring complex reasoning, as illustrated in Figure 6.

### B.6. Ablation Study of Language Consistency Reward

To study the impact of the Language Consistency (LC) Reward, we conduct an ablation experiment on DeepSeek-R1-Distill-Qwen-7B. This model uses the same cold start data as DeepSeek-R1