tricks

by David Oniani

Contents

1 Trie - A Prefix Tree

There are many ways to constuct a trie. We will use a hash map based approach. We first create a `Node` class.

```
class TrieNode:
    def __init__(self) -> None:
        """Initialize the `TrieNode` class."""
        self._children: dict[str, TrieNode] = {}
        self._isendofword: bool = False
```

We can now design the trie data structure. The initializer will only contain a root of the trie.

```
class Trie:
    def __init__(self) -> None:
        """Initialize the `Trie` class."""
        self._root = TrieNode()
```

Now, the first method to implement is the insertion operation. If we cannot insert a word into a trie, we are not going to be able to do much with it. The algorithm is very straightforward. One iterates over all characters in the word to be inserted and checks if the characters is in the children of the current node. If it is, follow the path of that node. If not, create a new node at that position and once again, follow that road.

```
def insert(self, word: str) -> None:
    """Initialize the `Trie` class."""
    ptr: Node = self._root
    for char in word:
        if char not in ptr._children:
            ptr._children[char] = TrieNode()
        ptr = ptr._children[char]

    ptr._isendofword = True
```

One of the primary features of the trie is to be able to search for a word. The implementation of this algorithm is shown below.

```
    def search(self, word: str) -> bool:
        """Check if a trie contains a given word."""
        ptr: Node = self._root
        for char in word:
            if char not in ptr._children:
                return False
            ptr = ptr._children[char]

        return ptr._isendofword
```

Similarly, searching a prefix is also very important. It looks almost exactly the same as the code for searching a word.

```
    def prefix(self, word: str) -> bool:
        """Check if a trie contains a given prefix."""
        ptr: Node = self._root
        for char in word:
            if char not in ptr._children:
                return False
            ptr = ptr._children[char]

        return True
```

2 Merge Sort

Merge sort is one of the simplest sorting algorithms that scales very well. It relies on two operations - `merge` and `sort`.

```
        def merge(arr1: list[int], arr2: list[int]) -> list[int]:
            """Merge two sorted lists."""
            res: list[int] = []

            i: int = 0
            j: int = 0
            while i <= len(arr1) - 1 and j <= len(arr2) - 1:
                if arr1[i] <= arr2[j]:
                    res.append(arr1[i])
                    i += 1
                else:
                    res.append(arr2[j])
                    j += 1

            while i <= len(arr1) - 1:
                res.append(arr1[i])
                i += 1

            while j <= len(arr2) - 1:
                res.append(arr2[j])
                j += 1

            return res
```

We now need to implement the sort function.

```
def sort(arr: list[int]) -> list[int]:
    """Performs a merge sort."""
    if len(nums) <= 1:
        return nums

    mid: int = len(arr) // 2
    left: list[int] = sort(arr[:mid])
    right: list[int] = sort(arr[mid:])

    return merge(left, right)
```