

# Problem 642: Design Search Autocomplete System

## Problem Information

Difficulty: **Hard**

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character

'#'

).

You are given a string array

`sentences`

and an integer array

`times`

both of length

`n`

where

`sentences[i]`

is a previously typed sentence and

times[i]

is the corresponding number of times the sentence was typed. For each input character except

'#'

, return the top

3

historical hot sentences that have the same prefix as the part of the sentence already typed.

Here are the specific rules:

The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.

The returned top

3

hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the same hot degree, use ASCII-code order (smaller one appears first).

If less than

3

hot sentences exist, return as many as you can.

When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Implement the

AutocompleteSystem

class:

`AutocompleteSystem(String[] sentences, int[] times)`

Initializes the object with the

`sentences`

and

`times`

arrays.

`List<String> input(char c)`

This indicates that the user typed the character

`c`

.

Returns an empty array

`[]`

if

`c == '#'`

and stores the inputted sentence in the system.

Returns the top

3

historical hot sentences that have the same prefix as the part of the sentence already typed. If there are fewer than

3

matches, return them all.

Example 1:

Input

```
["AutocompleteSystem", "input", "input", "input", "input"] [[["i love you", "island", "iroman", "i love leetcode"], [5, 3, 2, 2]], ["i"], [" "], ["a"], ["#"]]
```

Output

```
[null, ["i love you", "island", "i love leetcode"], ["i love you", "i love leetcode"], [], []]
```

Explanation

AutocompleteSystem obj = new AutocompleteSystem(["i love you", "island", "iroman", "i love leetcode"], [5, 3, 2, 2]); obj.input("i"); // return ["i love you", "island", "i love leetcode"]. There are four sentences that have prefix "i". Among them, "ironman" and "i love leetcode" have same hot degree. Since ' ' has ASCII code 32 and 'r' has ASCII code 114, "i love leetcode" should be in front of "ironman". Also we only need to output top 3 hot sentences, so "ironman" will be ignored. obj.input(" "); // return ["i love you", "i love leetcode"]. There are only two sentences that have prefix "i ". obj.input("a"); // return []. There are no sentences that have prefix "i a". obj.input("#"); // return []. The user finished the input, the sentence "i a" should be saved as a historical sentence in system. And the following input will be counted as a new search.

Constraints:

n == sentences.length

n == times.length

1 <= n <= 100

1 <= sentences[i].length <= 100

1 <= times[i] <= 50

c

is a lowercase English letter, a hash

'#'

, or space

''

.

Each tested sentence will be a sequence of characters

c

that end with the character

'#'

.

Each tested sentence will have a length in the range

[1, 200]

.

The words in each input sentence are separated by single spaces.

At most

5000

calls will be made to

input

.

## Code Snippets

### C++:

```
class AutocompleteSystem {
public:
    AutocompleteSystem(vector<string>& sentences, vector<int>& times) {

    }

    vector<string> input(char c) {

    }
};

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem* obj = new AutocompleteSystem(sentences, times);
 * vector<string> param_1 = obj->input(c);
 */
```

### Java:

```
class AutocompleteSystem {

    public AutocompleteSystem(String[] sentences, int[] times) {

    }

    public List<String> input(char c) {

    }
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem obj = new AutocompleteSystem(sentences, times);
 * List<String> param_1 = obj.input(c);
 */
```

### Python3:

```

class AutocompleteSystem:

    def __init__(self, sentences: List[str], times: List[int]):

    def input(self, c: str) -> List[str]:

# Your AutocompleteSystem object will be instantiated and called as such:
# obj = AutocompleteSystem(sentences, times)
# param_1 = obj.input(c)

```

## Python:

```

class AutocompleteSystem(object):

    def __init__(self, sentences, times):
        """
        :type sentences: List[str]
        :type times: List[int]
        """

    def input(self, c):
        """
        :type c: str
        :rtype: List[str]
        """

# Your AutocompleteSystem object will be instantiated and called as such:
# obj = AutocompleteSystem(sentences, times)
# param_1 = obj.input(c)

```

## JavaScript:

```

/**
 * @param {string[]} sentences
 * @param {number[]} times
 */
var AutocompleteSystem = function(sentences, times) {

```

```

};

/**
 * @param {character} c
 * @return {string[]}
 */
AutocompleteSystem.prototype.input = function(c) {

};

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * var obj = new AutocompleteSystem(sentences, times)
 * var param_1 = obj.input(c)
 */

```

### TypeScript:

```

class AutocompleteSystem {
  constructor(sentences: string[], times: number[]) {

  }

  input(c: string): string[] {

  }
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * var obj = new AutocompleteSystem(sentences, times)
 * var param_1 = obj.input(c)
 */

```

### C#:

```

public class AutocompleteSystem {

  public AutocompleteSystem(string[] sentences, int[] times) {

  }

}

```

```

public IList<string> Input(char c) {

}

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem obj = new AutocompleteSystem(sentences, times);
 * IList<string> param_1 = obj.Input(c);
 */

```

**C:**

```

typedef struct {

} AutocompleteSystem;

AutocompleteSystem* autocompleteSystemCreate(char** sentences, int
sentencesSize, int* times, int timesSize) {

}

char** autocompleteSystemInput(AutocompleteSystem* obj, char c, int* retSize)
{

}

void autocompleteSystemFree(AutocompleteSystem* obj) {

}

/**
 * Your AutocompleteSystem struct will be instantiated and called as such:
 * AutocompleteSystem* obj = autocompleteSystemCreate(sentences,
sentencesSize, times, timesSize);
 * char** param_1 = autocompleteSystemInput(obj, c, retSize);
 */

```

```
* autoCompleteSystemFree(obj);  
*/
```

## Go:

```
type AutocompleteSystem struct {  
  
}  
  
func Constructor(sentences []string, times []int) AutocompleteSystem {  
  
}  
  
func (this *AutocompleteSystem) Input(c byte) []string {  
  
}  
  
/**  
 * Your AutocompleteSystem object will be instantiated and called as such:  
 * obj := Constructor(sentences, times);  
 * param_1 := obj.Input(c);  
 */
```

## Kotlin:

```
class AutocompleteSystem(sentences: Array<String>, times: IntArray) {  
  
    fun input(c: Char): List<String> {  
  
    }  
  
}  
  
/**  
 * Your AutocompleteSystem object will be instantiated and called as such:  
 * var obj = AutocompleteSystem(sentences, times)  
 * var param_1 = obj.input(c)  
 */
```

## Swift:

```
class AutocompleteSystem {

    init(_ sentences: [String], _ times: [Int]) {

    }

    func input(_ c: Character) -> [String] {

    }
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * let obj = AutocompleteSystem(sentences, times)
 * let ret_1: [String] = obj.input(c)
 */
```

## Rust:

```
struct AutocompleteSystem {

}

/**
 * `&self` means the method takes an immutable reference.
 * If you need a mutable reference, change it to `&mut self` instead.
 */
impl AutocompleteSystem {

    fn new(sentences: Vec<String>, times: Vec<i32>) -> Self {

    }

    fn input(&self, c: char) -> Vec<String> {

    }
}

/**
```

```
* Your AutocompleteSystem object will be instantiated and called as such:
* let obj = AutocompleteSystem::new(sentences, times);
* let ret_1: Vec<String> = obj.input(c);
*/
```

## Ruby:

```
class AutocompleteSystem

  =begin
  :type sentences: String[]
  :type times: Integer[]
  =end
  def initialize(sentences, times)

  end

  =begin
  :type c: Character
  :rtype: String[]
  =end
  def input(c)

  end

end

# Your AutocompleteSystem object will be instantiated and called as such:
# obj = AutocompleteSystem.new(sentences, times)
# param_1 = obj.input(c)
```

## PHP:

```
class AutocompleteSystem {
    /**
     * @param String[] $sentences
     * @param Integer[] $times
     */
    function __construct($sentences, $times) {
```

```

}

/**
 * @param String $c
 * @return String[]
 */
function input($c) {

}
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * $obj = AutocompleteSystem($sentences, $times);
 * $ret_1 = $obj->input($c);
 */

```

#### Dart:

```

class AutocompleteSystem {

  AutocompleteSystem(List<String> sentences, List<int> times) {

  }

  List<String> input(String c) {

  }
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem obj = AutocompleteSystem(sentences, times);
 * List<String> param1 = obj.input(c);
 */

```

#### Scala:

```

class AutocompleteSystem(_sentences: Array[String], _times: Array[Int]) {

  def input(c: Char): List[String] = {

```

```

}

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * val obj = new AutocompleteSystem(sentences, times)
 * val param_1 = obj.input(c)
 */

```

## Elixir:

```

defmodule AutocompleteSystem do
  @spec init_(sentences :: [String.t], times :: [integer]) :: any
  def init_(sentences, times) do

  end

  @spec input(c :: char) :: [String.t]
  def input(c) do

  end
end

# Your functions will be called as such:
# AutocompleteSystem.init_(sentences, times)
# param_1 = AutocompleteSystem.input(c)

# AutocompleteSystem.init_ will be called before every test case, in which
# you can do some necessary initializations.

```

## Erlang:

```

-spec autocomplete_system_init_(Sentences :: [unicode:unicode_binary()],
Times :: [integer()]) -> any().
autocomplete_system_init_(Sentences, Times) ->
.

-spec autocomplete_system_input(C :: char()) -> [unicode:unicode_binary()].
autocomplete_system_input(C) ->
.

```

```
%% Your functions will be called as such:
%% autocomplete_system_init_(Sentences, Times),
%% Param_1 = autocomplete_system_input(C),

%% autocomplete_system_init_ will be called before every test case, in which
you can do some necessary initializations.
```

### Racket:

```
(define autocomplete-system%
(class object%
  (super-new)

  ; sentences : (listof string?)
  ; times : (listof exact-integer?)
  (init-field
   sentences
   times)

  ; input : char? -> (listof string?)
  (define/public (input c)
    )))

;; Your autocomplete-system% object will be instantiated and called as such:
;; (define obj (new autocomplete-system% [sentences sentences] [times
times]))
;; (define param_1 (send obj input c))
```

## Solutions

### C++ Solution:

```
/*
 * Problem: Design Search Autocomplete System
 * Difficulty: Hard
 * Tags: array, string, hash, sort, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
```

```

* Space Complexity: O(n) for hash map
*/

class AutocompleteSystem {
public:
    AutocompleteSystem(vector<string>& sentences, vector<int>& times) {

    }

    vector<string> input(char c) {

    }
};

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem* obj = new AutocompleteSystem(sentences, times);
 * vector<string> param_1 = obj->input(c);
 */

```

## Java Solution:

```

/**
 * Problem: Design Search Autocomplete System
 * Difficulty: Hard
 * Tags: array, string, hash, sort, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class AutocompleteSystem {

    public AutocompleteSystem(String[] sentences, int[] times) {

    }

    public List<String> input(char c) {

    }

}

```

```

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem obj = new AutocompleteSystem(sentences, times);
 * List<String> param_1 = obj.input(c);
 */

```

### Python3 Solution:

```

"""
Problem: Design Search Autocomplete System
Difficulty: Hard
Tags: array, string, hash, sort, search, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class AutocompleteSystem:

    def __init__(self, sentences: List[str], times: List[int]):

    def input(self, c: str) -> List[str]:
        # TODO: Implement optimized solution
        pass

```

### Python Solution:

```

class AutocompleteSystem(object):

    def __init__(self, sentences, times):
        """
        :type sentences: List[str]
        :type times: List[int]
        """

    def input(self, c):

```

```

"""
:type c: str
:rtype: List[str]
"""

# Your AutocompleteSystem object will be instantiated and called as such:
# obj = AutocompleteSystem(sentences, times)
# param_1 = obj.input(c)

```

### JavaScript Solution:

```

/**
 * Problem: Design Search Autocomplete System
 * Difficulty: Hard
 * Tags: array, string, hash, sort, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

/**
 * @param {string[]} sentences
 * @param {number[]} times
 */
var AutocompleteSystem = function(sentences, times) {

};

/**
 * @param {character} c
 * @return {string[]}
 */
AutocompleteSystem.prototype.input = function(c) {

};

/**
 * Your AutocompleteSystem object will be instantiated and called as such:

```

```

* var obj = new AutocompleteSystem(sentences, times)
* var param_1 = obj.input(c)
*/

```

## TypeScript Solution:

```

/**
 * Problem: Design Search Autocomplete System
 * Difficulty: Hard
 * Tags: array, string, hash, sort, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class AutocompleteSystem {
  constructor(sentences: string[], times: number[]) {

  }

  input(c: string): string[] {

  }
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * var obj = new AutocompleteSystem(sentences, times)
 * var param_1 = obj.input(c)
 */

```

## C# Solution:

```

/*
 * Problem: Design Search Autocomplete System
 * Difficulty: Hard
 * Tags: array, string, hash, sort, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)

```

```

* Space Complexity: O(n) for hash map
*/

public class AutocompleteSystem {

    public AutocompleteSystem(string[] sentences, int[] times) {

    }

    public IList<string> Input(char c) {

    }

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem obj = new AutocompleteSystem(sentences, times);
 * IList<string> param_1 = obj.Input(c);
 */

```

## C Solution:

```

/*
 * Problem: Design Search Autocomplete System
 * Difficulty: Hard
 * Tags: array, string, hash, sort, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

typedef struct {

} AutocompleteSystem;

AutocompleteSystem* autocompleteSystemCreate(char** sentences, int

```

```

sentencesSize, int* times, int timesSize) {

}

char** autocompleteSystemInput(AutocompleteSystem* obj, char c, int* retSize)
{

}

void autocompleteSystemFree(AutocompleteSystem* obj) {

}

/**
 * Your AutocompleteSystem struct will be instantiated and called as such:
 * AutocompleteSystem* obj = autocompleteSystemCreate(sentences,
sentencesSize, times, timesSize);
 * char** param_1 = autocompleteSystemInput(obj, c, retSize);

 * autocompleteSystemFree(obj);
 */

```

## Go Solution:

```

// Problem: Design Search Autocomplete System
// Difficulty: Hard
// Tags: array, string, hash, sort, search, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type AutocompleteSystem struct {

}

func Constructor(sentences []string, times []int) AutocompleteSystem {

}

```

```

func (this *AutocompleteSystem) Input(c byte) []string {

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * obj := Constructor(sentences, times);
 * param_1 := obj.Input(c);
 */

```

### Kotlin Solution:

```

class AutocompleteSystem(sentences: Array<String>, times: IntArray) {

    fun input(c: Char): List<String> {

    }

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * var obj = AutocompleteSystem(sentences, times)
 * var param_1 = obj.input(c)
 */

```

### Swift Solution:

```

class AutocompleteSystem {

    init(_ sentences: [String], _ times: [Int]) {

    }

    func input(_ c: Character) -> [String] {

    }

}

```

```

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * let obj = AutocompleteSystem(sentences, times)
 * let ret_1: [String] = obj.input(c)
 */

```

## Rust Solution:

```

// Problem: Design Search Autocomplete System
// Difficulty: Hard
// Tags: array, string, hash, sort, search, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct AutocompleteSystem {

}

/**
 * `&self` means the method takes an immutable reference.
 * If you need a mutable reference, change it to `&mut self` instead.
 */
impl AutocompleteSystem {

    fn new(sentences: Vec<String>, times: Vec<i32>) -> Self {

    }

    fn input(&self, c: char) -> Vec<String> {

    }
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * let obj = AutocompleteSystem::new(sentences, times);
 * let ret_1: Vec<String> = obj.input(c);
 */

```

## Ruby Solution:

```
class AutocompleteSystem

  =begin
  :type sentences: String[]
  :type times: Integer[]
  =end
  def initialize(sentences, times)

  end

  =begin
  :type c: Character
  :rtype: String[]
  =end
  def input(c)

  end

end

# Your AutocompleteSystem object will be instantiated and called as such:
# obj = AutocompleteSystem.new(sentences, times)
# param_1 = obj.input(c)
```

## PHP Solution:

```
class AutocompleteSystem {
    /**
     * @param String[] $sentences
     * @param Integer[] $times
     */
    function __construct($sentences, $times) {

    }

    /**
     * @param String $c
     * @return String[]
     */
}
```

```

*/
function input($c) {

}

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * $obj = AutocompleteSystem($sentences, $times);
 * $ret_1 = $obj->input($c);
 */

```

### Dart Solution:

```

class AutocompleteSystem {

  AutocompleteSystem(List<String> sentences, List<int> times) {

  }

  List<String> input(String c) {

  }

}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem obj = AutocompleteSystem(sentences, times);
 * List<String> param1 = obj.input(c);
 */

```

### Scala Solution:

```

class AutocompleteSystem(_sentences: Array[String], _times: Array[Int]) {

  def input(c: Char): List[String] = {

  }

}

```

```

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * val obj = new AutocompleteSystem(sentences, times)
 * val param_1 = obj.input(c)
 */

```

### Elixir Solution:

```

defmodule AutocompleteSystem do
  @spec init_(sentences :: [String.t], times :: [integer]) :: any
  def init_(sentences, times) do

  end

  @spec input(c :: char) :: [String.t]
  def input(c) do

  end
end

# Your functions will be called as such:
# AutocompleteSystem.init_(sentences, times)
# param_1 = AutocompleteSystem.input(c)

# AutocompleteSystem.init_ will be called before every test case, in which
# you can do some necessary initializations.

```

### Erlang Solution:

```

-spec autocomplete_system_init_(Sentences :: [unicode:unicode_binary()],
Times :: [integer()]) -> any().
autocomplete_system_init_(Sentences, Times) ->
.

-spec autocomplete_system_input(C :: char()) -> [unicode:unicode_binary()].
autocomplete_system_input(C) ->
.

%% Your functions will be called as such:
%% autocomplete_system_init_(Sentences, Times),

```

```
%% Param_1 = autocomplete_system_input(C),

%% autocomplete_system_init_ will be called before every test case, in which
you can do some necessary initializations.
```

### Racket Solution:

```
(define autocomplete-system%
  (class object%
    (super-new)

    ; sentences : (listof string?)
    ; times : (listof exact-integer?)
    (init-field
      sentences
      times)

    ; input : char? -> (listof string?)
    (define/public (input c)
      )))

;; Your autocomplete-system% object will be instantiated and called as such:
;; (define obj (new autocomplete-system% [sentences sentences] [times
times]))
;; (define param_1 (send obj input c))
```