Problem 808: Soup Servings

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You have two soups,

A

and

B

, each starting with

n

mL. On every turn, one of the following four serving operations is chosen

at random

, each with probability

0.25

independent

of all previous turns:

pour 100 mL from type A and 0 mL from type B pour 75 mL from type A and 25 mL from type B

pour 50 mL from type A and 50 mL from type B

pour 25 mL from type A and 75 mL from type B

Note:

There is no operation that pours 0 mL from A and 100 mL from B.

The amounts from A and B are poured

simultaneously

during the turn.

If an operation asks you to pour

more than

you have left of a soup, pour all that remains of that soup.

The process stops immediately after any turn in which

one of the soups

is used up.

Return the probability that A is used up

before

B, plus half the probability that both soups are used up in the

same turn

. Answers within

10

-5

of the actual answer will be accepted.

Example 1:

Input:

n = 50

Output:

0.62500

Explanation:

If we perform either of the first two serving operations, soup A will become empty first. If we perform the third operation, A and B will become empty at the same time. If we perform the fourth operation, B will become empty first. So the total probability of A becoming empty first plus half the probability that A and B become empty at the same time, is 0.25 * (1 + 1 + 0.5 + 0) = 0.625.

Example 2:

Input:

n = 100

Output:

0.71875

Explanation:

If we perform the first serving operation, soup A will become empty first. If we perform the second serving operations, A will become empty on performing operation [1, 2, 3], and both A and B become empty on performing operation 4. If we perform the third operation, A will become empty on performing operation [1, 2], and both A and B become empty on performing operation 3. If we perform the fourth operation, A will become empty on performing operation

1, and both A and B become empty on performing operation 2. So the total probability of A becoming empty first plus half the probability that A and B become empty at the same time, is 0.71875.

Constraints:

0 <= n <= 10

9

Code Snippets

C++:

```
class Solution {
public:
double soupServings(int n) {

}
};
```

Java:

```
class Solution {
public double soupServings(int n) {

}
}
```

Python3:

```
class Solution:
def soupServings(self, n: int) -> float:
```

Python:

```
class Solution(object):
def soupServings(self, n):
"""
:type n: int
:rtype: float
```

```
"""
```

JavaScript:

```
/**
* @param {number} n
* @return {number}
*/
var soupServings = function(n) {

};
```

TypeScript:

```
function soupServings(n: number): number {

};
```

C#:

```
public class Solution {
public double SoupServings(int n) {

}
}
```

C:

```
double soupServings(int n) {

}
```

Go:

```
func soupServings(n int) float64 {

}
```

Kotlin:

```
class Solution {
fun soupServings(n: Int): Double {
```

```
}
}
```

Swift:

```
class Solution {
func soupServings(_ n: Int) -> Double {

}
}
```

Rust:

```
impl Solution {
pub fn soup_servings(n: i32) -> f64 {

}
}
```

Ruby:

```
# @param {Integer} n
# @return {Float}
def soup_servings(n)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @return Float
*/
function soupServings($n) {

}
}
```

Dart:

```
class Solution {
double soupServings(int n) {

}
}
```

Scala:

```
object Solution {
def soupServings(n: Int): Double = {

}
}
```

Elixir:

```
defmodule Solution do
@spec soup_servings(n :: integer) :: float
def soup_servings(n) do

end
end
```

Erlang:

```
-spec soup_servings(N :: integer()) -> float().
soup_servings(N) ->
.
```

Racket:

```
(define/contract (soup-servings n)
(-> exact-integer? flonum?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Soup Servings
```

```
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
double soupServings(int n) {

}
};
```

Java Solution:

```
/**
* Problem: Soup Servings
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public double soupServings(int n) {

}
}
```

Python3 Solution:

```
"""
Problem: Soup Servings
Difficulty: Medium
Tags: dp, math

Approach: Dynamic programming with memoization or tabulation
```

```
Time Complexity: O(n * m) where n and m are problem dimensions
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def soupServings(self, n: int) -> float:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def soupServings(self, n):
"""
:type n: int
:rtype: float
"""
```

JavaScript Solution:

```
/**
* Problem: Soup Servings
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} n
* @return {number}
*/
var soupServings = function(n) {

};
```

TypeScript Solution:

```
/**
* Problem: Soup Servings
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

function soupServings(n: number): number {

};
```

C# Solution:

```
/*
* Problem: Soup Servings
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public double SoupServings(int n) {

}
}
```

C Solution:

```
/*
* Problem: Soup Servings
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

double soupServings(int n) {

}
```

Go Solution:

```
// Problem: Soup Servings
// Difficulty: Medium
// Tags: dp, math
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
// Space Complexity: O(n) or O(n * m) for DP table

func soupServings(n int) float64 {

}
```

Kotlin Solution:

```
class Solution {
fun soupServings(n: Int): Double {

}
}
```

Swift Solution:

```
class Solution {
func soupServings(_ n: Int) -> Double {

}
}
```

Rust Solution:

```
// Problem: Soup Servings
// Difficulty: Medium
// Tags: dp, math
```

```
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn soup_servings(n: i32) -> f64 {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @return {Float}
def soup_servings(n)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @return Float
*/
function soupServings($n) {

}
}
```

Dart Solution:

```
class Solution {
double soupServings(int n) {

}
}
```

Scala Solution:

```
object Solution {
def soupServings(n: Int): Double = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec soup_servings(n :: integer) :: float
def soup_servings(n) do

end
end
```

Erlang Solution:

```
-spec soup_servings(N :: integer()) -> float().
soup_servings(N) ->
.
```

Racket Solution:

```
(define/contract (soup-servings n)
(-> exact-integer? flonum?)
)
```