Soup Servings

There are two types of soup: type A and type B. Initially we have `N` ml of each type of soup. There are four kinds of operations:

1. Serve 100 ml of soup A and 0 ml of soup B
2. Serve 75 ml of soup A and 25 ml of soup B
3. Serve 50 ml of soup A and 50 ml of soup B
4. Serve 25 ml of soup A and 75 ml of soup B

When we serve some soup, we give it to someone and we no longer have it. Each turn, we will choose from the four operations with equal probability 0.25. If the remaining volume of soup is not enough to complete the operation, we will serve as much as we can. We stop once we no longer have some quantity of both types of soup.

Note that we do not have the operation where all 100 ml's of soup B are used first.

Return the probability that soup A will be empty first, plus half the probability that A and B become empty at the same time.

```
Example:
Input: N = 50
Output: 0.625
Explanation:
If we choose the first two operations, A will become empty first. For the third opera
tion, A and B will become empty at the same time. For the fourth operation, B will be
come empty first. So the total probability of A becoming empty first plus half the pr
obability that A and B become empty at the same time, is 0.25 * (1 + 1 + 0.5 + 0) = 0
.625.
```

Notes:

- `0 <= N <= 10^9`.
- Answers within `10^-6` of the true value will be accepted as correct.

Solution 1

First, 25ml is annoying
The decription is very difficult to understand. And all `25ml` just make it worse.
when I finally figure it out, I consider only how many servings left in A and B.
`1 serving = 25ml`. (Well, it works similar to your milk powder or protin powder.)
If the left part is not enough for one serving, it is considered as one serving as well.
I have to say **this process is necessary**. Maybe some other solution get accepted with `map`, they are just lucky because the test case is weak and incompleted.
Becasue `5000 * 5000 * Double` is too big and memory limit will be exceeded.
If a solution don't do this process, it should be wrong.

Second, DP or recursion with memory
Now it's easy to solve this problem.
`f(a,b)` means the result probability for a ml of soup A and b ml of soup B.
`f(a-4,b)` means that we take the first operation: Serve 100 ml of soup A and 0 ml of soup B. `f(a-3,b-1), f(a-2,b-2), f(a-1,b-3)` are other 3 operations.
The condition `a <= 0 and b <= 0` means that we run out of soup A and B at the same time, so we should return a probability of `0.5`, which is half of `1.0`.
The same idea for other two conditions.
I cached the process as we do for Fibonacci sequence. It calculate every case for only once and it can be reused for every test case. No worries for TLE.

Third, take the hint for big N
"Note that we do not have the operation where all 100 ml's of soup B are used first. "
It's obvious that A is easier to be empty than B. And when `N` gets bigger, we have less chance to run out of B first.
So as `N` increases, our result increases and it gets closer to 100 percent = 1.

"Answers within 10^-6 of the true value will be accepted as correct."
Now it's obvious that when `N` is big enough, result is close enough to 1 and we just need to return 1.
When I incresed the value of `N`, I find that:
When `N = 4800`, the `result = 0.999994994426`
When `N = 4801`, the `result = 0.0.999995382315`
So if N>= 4800, just return 1 and it will be enough.

Complexity
O(N^2) for time and space.
But we restrict the value of N to be smaller than 4800.

C++:

```
    double memo[200][200];
    double soupServings(int N) {
        return N >= 4800 ?  1.0 : f((N + 24) / 25, (N + 24) / 25);
    }
    double f(int a, int b) {
        if (a <= 0 && b <= 0) return 0.5;
        if (a <= 0) return 1;
        if (b <= 0) return 0;
        if (memo[a][b] > 0) return memo[a][b];
        memo[a][b] = 0.25 * (f(a-4,b)+f(a-3,b-1)+f(a-2,b-2)+f(a-1,b-3));
        return memo[a][b];
    }
```

Java:

```
    double[][] memo = new double[200][200];
    public double soupServings(int N) {
        return N >= 4800 ?  1.0 : f((N + 24) / 25, (N + 24) / 25);
    }

    public double f(int a, int b) {
        if (a <= 0 && b <= 0) return 0.5;
        if (a <= 0) return 1;
        if (b <= 0) return 0;
        if (memo[a][b] > 0) return memo[a][b];
        memo[a][b] = 0.25 * (f(a - 4, b) + f(a - 3, b - 1) + f(a - 2, b - 2) + f(a
- 1, b - 3)));
        return memo[a][b]
    }
```

Python:

```
class Solution(object):
    memo = {}
    def soupServings(self, N):
        if N > 4800: return 1
        def f(a, b):
            if (a, b) in self.memo: return self.memo[a, b]
            if a <= 0 and b <= 0: return 0.5
            if a <= 0: return 1
            if b <= 0: return 0
            self.memo[(a, b)] = 0.25 * (f(a - 4, b) + f(a - 3, b - 1) + f(a - 2, b
- 2) + f(a - 1, b - 3))
            return self.memo[(a, b)]
        N = math.ceil(N / 25.0)
        return f(N, N)
```

written by lee215 original link here

Solution 2

The most important hint in this problem for me is `"Answers within 10^-6 of the true value will be accepted as correct."`.
And when I get timed out or runtime error with my DP, I tried to print out results of each call for each N.

They are monotonically increasing, and getting closer and closer to 1.
If you print it out, you will observe that when it is as large as 5551 the gap between the result and 1 is less than 10^-6.

Note that, the memoization is necessary, otherwise it will get TLE.

```
class Solution(object):
    def soupServings(self, N):
        if N>=5551: return 1
        self.dd=collections.defaultdict(float)
        return self.sub(N,N)

    def sub(self, an, bn):
        if an<=0 and bn<=0: return 0.5
        if an<=0: return 1
        if bn<=0: return 0
        if (an,bn) in self.dd: return self.dd[(an,bn)]
        c1=self.sub(an-100, bn)
        c2=self.sub(an-75, bn-25)
        c3=self.sub(an-50, bn-50)
        c4=self.sub(an-25, bn-75)
        self.dd[(an,bn)]=0.25*sum([c1,c2,c3,c4])
        return self.dd[(an,bn)]
```

written by licaiuu original link here

Solution 3

DP: much faster

```
class Solution {
public:
  //Using DP. much more faster
  vector<pair<int,int>> dir = {{-4, 0}, {-3, -1}, {-2, -2}, {-1, -3}};
  double soupServings(int N) {
   N = ceil(N/25.0);
   //cout<<N<<endl;
   //everything is divded 25=> save space.
   if(N > 500) return 1.0;
   vector<vector<double> > dp(N+1, vector<double>(N+1, 0.0));
   for(int i = 0; i < N+1; ++i){
     for(int j = 0; j < N+1; ++j){
       if(i == 0 && j > 0){
         dp[i][j] = 1.0;
       }else if(i == 0 && j == 0){
         dp[i][j] = 0.5;//1.0*1/2 = 0.5
       }else if(i > 0 && j > 0){
         for(auto d : dir){
           dp[i][j] += 0.25 * dp[max(d.first + i, 0)][max(d.second + j, 0)];
         }
       }
     }
   }
   return dp[N][N];
  }
};
```

My first idea: Using unorderede_map and DFS. Accepted, 50ms around.

```
class Solution {
 public:
  vector<pair<int,int>> dir = {{-100, 0}, {-75, -25}, {-50, -50}, {-25, -75}};
  unordered_map<string, pair<double, double>> mp;//"A,B": (prob1_Aempty: prob2_both
)
  double soupServings(int N) {
    if(N > 12500) return 1.0;//save lots of time!!!!
    auto p = search({N, N}, 1.0);

    return p.first + 0.5 * p.second;
  }

  pair<long double, long double> search(pair<int, int> Vol, long double prob){
    string cur_status = to_string(Vol.first) + ',' + to_string(Vol.second);
    pair<double, double> res = {0.0, 0.0};
    if(mp.find(cur_status) != mp.end()){
      res = mp[cur_status];
    }else if(Vol.first == 0 && Vol.second > 0){
      res = {prob, 0.0};
    }else if(Vol.first == 0 && Vol.second == 0){
      res = {0.0, prob};
    }else if(Vol.first > 0 && Vol.second == 0){
      res = {0.0, 0.0};
    }else{
      for(auto d : dir){
        pair<int, int> tmp_p = {max(d.first + Vol.first,0), max(d.second + Vol.seco
nd,0)};
        string tmp_status = to_string(tmp_p.first) + ',' + to_string(tmp_p.second);

        auto tmp = search(tmp_p, prob*0.25);
        res.first += tmp.first;
        res.second += tmp.second;
      }
      mp[cur_status] = res;
    }
    return res;
  }
 };
```

written by jasonshieh original link here