1 Lucky Dip - Round A - Kick Start 2018

1.1 Meaning of the problem

Given a array $V[]$ of length N. We need to randomly choose an element of the array. We know the value of each element of the array beforehand. If we think the element we choose is not large enough, we can choose again. The largest times we can choose is K + 1. We need to calculate the expected value of the element we at last get.

1.2 Keyword

Dynamic Programming

1.3 Idea

We use an array $dp[]$. $dp[i]$ means the expected value of the element we get when we can choose i + 1 times.

Apparently when i = 0, we can only choose only. $dp[0] = \sum_{j=1}^{N-1} V[j]/N$.

We then consider $dp[1]$. If we get something $\geq dp[0]$ at the first time we choose, we keep that and don't choose again. But if we get something $< dp[0]$, we choose again and the expectation of this choice is $dp[0]$. So we want to find the smallest index x in ascendingly sorted array $V[]$ and $V[x] \geq dp[0]$.

Then $dp[1] = \dfrac{(x * dp[0] + \sum_{j=x}^{N-1} V[j])}{N}$.

Simillary, when we consider $dp[i]$. If we get something $\geq dp[i-1]$ at the first time we choose, we keep that and don't choose again. But if we get something $< dp[i-1]$, we choose again and the expectation of this choice is $dp[i-1]$. So we want to find the smallest index x in ascendingly sorted array $V[]$ and $V[x] \geq dp[i-1]$.

Then $dp[i] = \dfrac{(x * dp[i-1] + \sum_{j=x}^{N-1} V[j])}{N}$.

1.4 Usages

1. sort(V, V + N): sort array V of length N.
2. $upper\_bound(V, V + N, X)$: return the index of the first element that is > X.
3. Input and output

```
int I;
double F = 1;
scanf("%d", &I);
print("%d %f", I, F);
```

1.5 Optimization

1. Use Prefix summation to calculate $\sum_{j=x}^{N-1} V[j]$.

We use an array sum. $sum[i]$ means $\sum_{j=0}^{i} V[j]$.
When x - 1 $\geq$ 0,

$$\sum_{j=x}^{N-1} V[j] = sum[N-1] - sum[x-1]$$

When x - 1 $<$ 0, i.e. x $=$ 0,

$$\sum_{j=x}^{N-1} V[j] = sum[N-1]$$

1.6 Code

```
// 1.1.cpp
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<algorithm>
using namespace std;

int V[20001];
long long sum[20001];
double dp[50001];

int main() {
  int T; // There is totally T cases.
  scanf("%d", &T);
  int caseNum = 1;
  while (caseNum <= T) {
    int K;// Totally we can choose K + 1 times
    int N;// The length of the array is N
    scanf("%d%d", &N, &K);
    for (int i = 0; i < N; i++) {
      scanf("%d", &V[i]);
    }
    sort(V, V + N);// Sort the array V
    // Calculate the prefix summation
    for (int i = 0; i < N; i++) {
      if (i == 0) {
        sum[0] = V[0];
      }
      else {
        sum[i] = sum[i - 1] + V[i];
      }
    }
    // Calculate dp
    for (int i = 0; i <= K; i++) {
      if (i == 0) {
        dp[0] = (double)sum[N - 1] / N;
      }
      else {
          // Get the index of the first element
          // that is greater than dp[i - 1]
        int x = upper_bound(V, V + N, dp[i - 1]) - V;
```

```cpp
        long long summation = 0;
        if (x == 0)
          summation = sum[N - 1];
        else
          summation = sum[N - 1] - sum[x - 1];
        dp[i] = (double)(x * dp[i - 1] + summation) / N;
      }
    }
    printf("Case #%d: %f\n", caseNum, dp[K]);
    caseNum += 1;
  }
  return 0;
}
```

1.7 Adjustments

1. We can write *upper_bound* function on our own using binary search.

```cpp
// 1.2.cpp
int upperHelper(int arr[], int target, int length, int start, int end) {
  // We want to search in [start, end]
  if (start == end) {
    if (arr[start] > target)
      return start;
    else
      return start + 1;
  }
  int half = (start + end) / 2;
  if (arr[half] <= target) {
    return upperHelper(arr, target, length, half + 1, end);
  }
  else {
    return upperHelper(arr, target, length, start, half);
  }
}
int upper(int arr[], int target, int length) {
  return upperHelper(arr, target, length, 0, length - 1);
}
```

1.8 Errors

1. I used a type conversion that int $- >$ long long $- >$ int, which caused error.