**Code Jam 2016 - Round 2**

# Analysis: Red Tape Committee

## Red Tape Committee: Analysis

This problem poses two challenges: figuring out which sets of members to consider as possible committees, and calculating the tie probability for each committee. In the Small dataset, brute force will suffice for both of these subproblems. However, in the Large dataset, we will need more efficient methods for both of them.

### Who should we choose?

One might think that in order to create ties, we should choose from among the most "moderate" department members — that is, the ones with Yes probabilities closest to 0.5. In fact, the opposite is true! The best way to create a tie is to choose department members from one or both *extremes*. That is, we should choose the *M* (possibly zero) of the department members with the lowest Yes probabilities, and the **K** - *M* of the department members with the highest Yes probabilities. This makes sense intuitively; to take an extreme case, a committee of two members with Yes probabilities of 0.00 and 1.00 will always tie, whereas a committee of two members with Yes probabilities of 0.50 and 0.50 will tie only half the time. Experimentation bears this idea out. But how can we prove it?

Without loss of generality, let's sort the members in increasing order of Yes probability. Suppose that we have chosen a committee of these members that maximizes the tie probability. If there are multiple such committees, suppose that we have chosen the one that also minimizes the sum of the members' indexes in that sorted list.

We'll show that this set consists of the *M* (possibly zero) leftmost members and the **K** - *M* rightmost members, as described above. Suppose that there exist the following: a member X, who is in our set, and members Y and Z, who are not in our set, and that they are in the left to right order Y, X, Z. Fix all the other members and consider the tie probability as a function of member X's Yes probability. This is a linear function. If it has slope 0, then we can get an equally good set with a smaller sum of member indices by replacing X with Y. If it has slope > 0, we can get a better set by replacing X with Z. If it has slope < 0, we can get a better set by replacing X with Y. So X must not exist!

Therefore, we can try all values of *M* and consider only those committees. This linear search adds a multiplier of O(**K**) to the running time of the calculation of tie probabilities. The one-time sort also adds a single O(**N** log **N**) term.

### How likely is a tie?

For a large committee, we cannot explicitly consider all $2^{\mathbf{N}}$ possible voting outcomes. Many of these outcomes are very similar, and we would do a lot of redundant work. This is an ideal situation for dynamic programming.

Let's build a table in which the columns represent the committee members, the rows represent the total number of Yes votes so far, and the numbers in the cells measure the probability of being in the situation represented by that row and column. We start with a 1.00 in the upper left cell, which represents the situation before anyone has voted; there is a 100% chance that there will be no "Yes" votes at this stage. Let's consider a committee with Yes probabilities of 0.10, 0.20, 0.50, and 1.00. We will label the columns in that order (although the order does not matter).

| - | init | 0.10 | 0.20 | 0.50 | 1.00 |
|---|---|---|---|---|---|
| **0** | 1.00 | ---- | ---- | ---- | ---- |
| **1** | ---- | ---- | ---- | ---- | ---- |
| **2** | ---- | ---- | ---- | ---- | ---- |
| **3** | ---- | ---- | ---- | ---- | ---- |
| **4** | ---- | ---- | ---- | ---- | ---- |

When the first member votes, either the vote will be "Yes" with 10% probability (and we will have one Yes vote), or "No" with 90% probability (and we will have zero Yes votes). So the 1.00 value gets split up among two cells in the next column: the "0 Yes votes after 1 member has voted" and "1 Yes vote after 1 member has voted" cells.

| - | init | 0.10 | 0.20 | 0.50 | 1.00 |
|---|---|---|---|---|---|
| **0** | 1.00 | 0.90 | ---- | ---- | ---- |
| **1** | ---- | 0.10 | ---- | ---- | ---- |
| **2** | ---- | ---- | ---- | ---- | ---- |
| **3** | ---- | ---- | ---- | ---- | ---- |
| **4** | ---- | ---- | ---- | ---- | ---- |

Let's look at the "0 Yes votes after 1 member has voted" cell, which represents 90% of all possible situations after the first member has voted. That probability will feed into two of the cells in the next column: the one just to the right, and the one just below that. Since the second member has an 80% probability of voting No, 80% of that 90% possibility space branches off to the "0 Yes votes after 2 members have voted" cell. The other 20% of that 90% branches off to the "1 Yes vote after 2 members have voted" cell.

| - | init | 0.10 | 0.20 | 0.50 | 1.00 |
|---|---|---|---|---|---|
| **0** | 1.00 | 0.90 | 0.72 | ---- | ---- |
| **1** | ---- | 0.10 | 0.18 | ---- | ---- |
| **2** | ---- | ---- | ---- | ---- | ---- |
| **3** | ---- | ---- | ---- | ---- | ---- |
| **4** | ---- | ---- | ---- | ---- | ---- |

And now for the "1 Yes vote after 1 member has voted" cell, which represents 10% of all possible situations after the first member has voted. Again, that probability will feed into the right and down-and-right neighboring cells in the next column. Note that we add 0.08 to the existing value of 0.18 in the "1 Yes vote after 2 members have voted" cell; there are multiple ways of getting to that cell. The power of dynamic programming is that it merges separate possibilities like this and lets us consider them together going forward; this prevents an exponential increase in the number of possibilities.

| - | init | 0.10 | 0.20 | 0.50 | 1.00 |
|---|---|---|---|---|---|
| **0** | 1.00 | 0.90 | 0.72 | ---- | ---- |
| **1** | ---- | 0.10 | 0.26 | ---- | ---- |
| **2** | ---- | ---- | 0.02 | ---- | ---- |
| **3** | ---- | ---- | ---- | ---- | ---- |
| **4** | ---- | ---- | ---- | ---- | ---- |

Continuing in this way, we can fill in the whole table. Note that every column sums to 1, as expected.

| - | init | 0.10 | 0.20 | 0.50 | 1.00 |
|---|---|---|---|---|---|
| **0** | 1.00 | 0.90 | 0.72 | 0.36 | 0.00 |
| **1** | ---- | 0.10 | 0.26 | 0.49 | 0.36 |
| **2** | ---- | ---- | 0.02 | 0.14 | 0.49 |

| | | | | | |
|---|---|---|---|---|---|
| **3** | ---- | ---- | ---- | 0.01 | 0.14 |
| **4** | ---- | ---- | ---- | ---- | 0.01 |

The tie probability is the value in the "2 Yes votes after 4 members have voted" cell: 0.49. We could have optimized this further by not considering any rows below the number of Yes votes needed for a tie. In practice, in problems like this, one should store the logarithms of probabilities instead of the actual values, which can become small enough for floating-point precision errors to matter.

The number of calculations in this method is proportional to the dimensions of the table, each of which is proportional to **K**, so the running time of this part is O($\mathbf{K}^2$). Combining that with the O(**K**) method of selecting committees, the overall running time of our approach is O($\mathbf{K}^3$) + O(**N** log **N**). Since **K** cannot exceed **N**, and **N** cannot exceed 200 for the Large, this is fast enough, albeit not optimal. (For example, we could do a ternary search on the value of *M* mentioned above, instead of a linear search.)