Detailed Algorithm Analysis: Collection Problem

The problem is as follows: We wish to collect all the items in a set, {1, 2, 3, …, n}. Unfortunately, we can only buy "random" items. Each time we buy an item, we have no control over which one we get. We simply have an equal chance or receiving each item each time we choose to buy one.

Given this scenario, the problem is, how many items do we have to buy on average, before we expect to get the entire set?

The first item that will help us is the linearity of expectation.

In particular, if we want to find the expected time of completing task A, followed by task B, followed by task C, etc. when we can only move on after completing the previous task, we can simply add up the expected amount of times for completing each task separately and add. This intuitively makes sense to us, and mathematically can be proven as well! (Math and intuition don't always agree, but we're lucky on this particular problem.)

Each task that we have is of the following form:

Given that I already own k distinct items out of the set of n, how many items do I expect to buy before I receive my k+1 item?

When k is 0, it’s clear to see that we’ll buy EXACTLY 1 item before we get a new one. From there, it’s less clear.

Expected Number of items to buy to get ONE new item

Let X equal the expected number of items we must buy to obtain our k+1 distinct item.

If we are lucky, which will happen (n – k)/n of the time, we will ONLY have to buy 1 item.

The other k/n of the time, we will have bought one item and be no better off than if we had bought none. In this case, we expect to buy X+1 items, the X being the average number we expect to buy from this point on, and the 1 representing this repeated item we just bought.

This gives us the following equation:

$$X = (n - k)/n*1 + k/n*(X+1)$$

The idea is that these are two separate expressions for the same expectation. The LHS is simply the variable we designated for this value while the RHS is broken down into the two distinct probabilities of what happens when we buy our first item after we have k distinct ones.

Now, solve this equation for X:

X = (n – k)/n*1 + k/n*(X+1)

X = (n – k)/n + k/n*X + k/n

X – k/n*X = (n – k + k)/n

X(1 – k/n) = n/n

X((n – k)/n) = 1

X = n/(n – k)

Thus, we can expect to buy n/(n-k) items to receive our k+1 item.

Solving the Problem

Thus, we know that we expect to buy 1 item to get our first new item, n/(n – 1) items to get our second item, n/(n – 2) items to get our third new item, etc.

Using the linearity of expectation, to figure out the total number of items we need to buy, we get the following sum:

n/n + n/(n – 1) + n/(n – 2) + n/(n – 3) + … + n/1

Factoring n out of this expression, we get:

n(1/n + 1/(n – 1) + 1/(n – 2) + 1/(n – 3) + … + 1/1)

The sum inside the parentheses is known as the Harmonic Sum.

A little bit of calculus can be used to show that this sum is very close to the value ln n. Thus, we find that the total number of items we must buy (on average) to get the whole set is very close to nln.

(In particular, we can represent the sum as n rectangles along the x-axis with width 1 and heights 1, ½, 1/3, etc. The sum of the areas of these rectangles equals the value of this sum. We can get an upper and lower bound on the area of these rectangles by drawing the functions f(x) = 1/x and f(x) = 1/(x+1) and seeing how they interact in relation to the rectangles. Getting the areas under these curves within the designated ranges is fairly easy using calculus and gives us good tight approximations to the sum.)

Code to Simulate the Process

```
int buyWholeSet(int n) {

    int numDistinct = 0;
    int numSteps = 0;
    int* gotIt = (int*)malloc(sizeof(int)*n);

    int i;
    for (i=0; i<n; i++)
        gotIt[i] = 0;

    while (numDistinct < n) {

       numSteps++;
       int item = ((1 << 15)*rand() + rand())%n;

       if (gotIt[item] == 0) {
          gotIt[item] = 1;
          numDistinct++;
       }
    }
```

```
    free(gotIt);
    return numSteps;
}


/*
Note: Since rand() only returns numbers upto
32767, in order to create random item numbers
beyond this, bitshifting is used to create a
random number that can approach 2 to the 30
power.
*/
```