Lab 4: Second Moment Estimation and Load Balancing
Due at 11:59PM on November, 28th as a PDF via websubmit.

Vidya Akavoor

November 27, 2016

Collaborators: Sreeja Keesara, Lauren DiSalvo
Sources:

You can enter the references by hand like this:

1. Piazza `https://piazza.com/class/is95ssbu4rq3t5?cid=434#`
2. Moment (mathematics) `https://en.wikipedia.org/wiki/Moment_(mathematics)`

Late days for this assignment: 0
Total late days this semester: 3 days left

Load Balancing

Problem 1:

```
def deterministicLoadBalancer(n):
    array = [0]*n
    for i in range(n):
        array[array.index(min(array))] += 1
    return array
```

Problem 2: The run time is n^2 because for each of the n shoppers, it needs to look at n lines.

Problem 3:

```
def loadBalancer(n):
    array = [0]*n
    for x in range(n):
        idx = rnd.randint(0,n-1)
        array[idx] += 1
    return max(array)
```

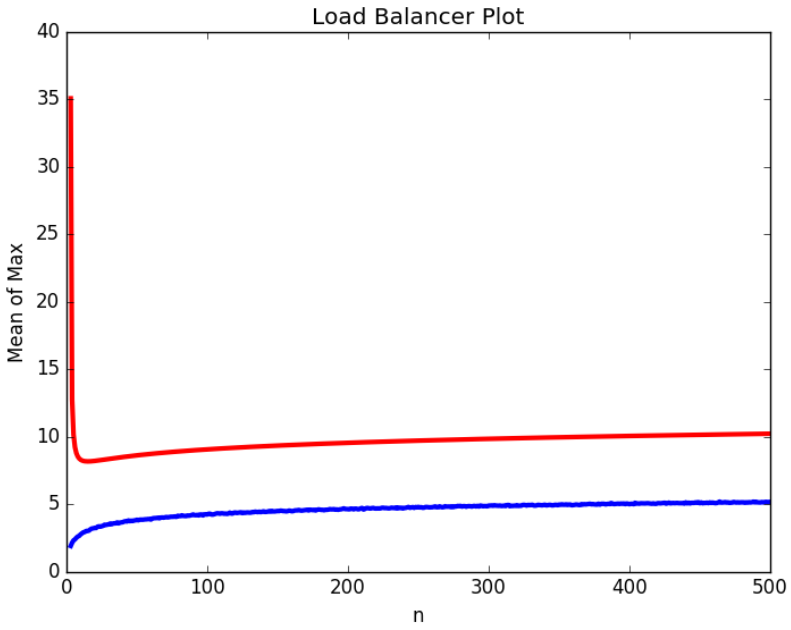
Problem 4: The run time is n because it goes through n shoppers once. It is faster than the deterministic one.

Problem 5: The trade off is the risk that all n shoppers get put in the same line.

Problem 6: The worst case (the longest line you could have) is n shoppers long. The probability of this happening is $n/(n^n)$ because there is a $1/n$ chance of one shopper being in a given line to the n for n shoppers, times n for n lines.

Problem 7:

```
def loadBalancerPlot():
    mean_list = []
    y_coord = []
    for i in range(3, 501):
        max_list = [0]*1000
        for j in range(1000):
            max_array = loadBalancer(i)
            max_list[j] = max_array
        mean_list += [np.mean(max_list)]
        y_coord += [(3*math.log(i))/(math.log(math.log(i)))]
    plt.figure(1)
    plt.plot(range(3,501), mean_list, linewidth = 3)
    plt.draw()
    x_coord = range(3,501)
    plt.plot(x_coord, y_coord, color = 'r', linewidth = 3)
    plt.xlabel("n")
    plt.ylabel("Mean of Max")
    plt.title("Load Balancer Plot")
    plt.show()
```



Problem 8:

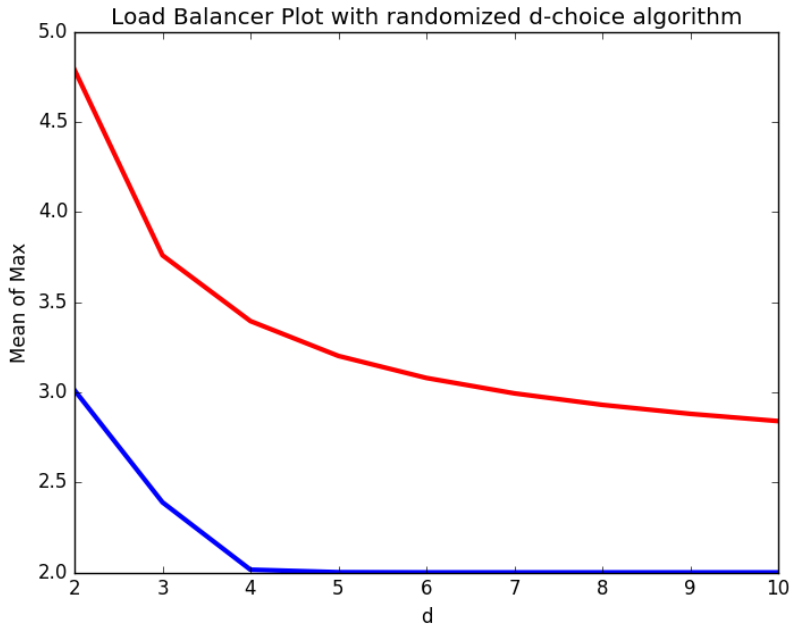
```
def loadBalancer2(n, d):
    array = [0]*n
```

```
    for x in range(n):
        idx = np.random.choice(array, d, replace = False)
        array[array.index(min(idx))] += 1
    return max(array)
```

Problem 9: Theoretically, if d and n were the same, we would get the same runtime as the deterministicLoadBalancer(n) function and the max should be 1 every time.

Problem 10:

```
def loadBalancerPlot2():
    mean_list = []
    y_coord = []
    for i in range(2, 11):
        max_list = [0]*1000
        for j in range(1000):
            max_array = loadBalancer2(1000, i)
            max_list[j] = max_array
        mean_list += [np.mean(max_list)]
        y_coord += [((math.log(math.log(1000)))/(math.log(i)))+2]
    plt.figure(1)
    plt.plot(range(2,11), mean_list, linewidth = 3)
    plt.draw()
    x_coord = range(2, 11)
    plt.plot(x_coord, y_coord, color = 'r', linewidth = 3)
    plt.xlabel("d")
    plt.ylabel("Mean of Max")
    plt.title("Load Balancer Plot with randomized d-choice algorithm")
    plt.show()
```



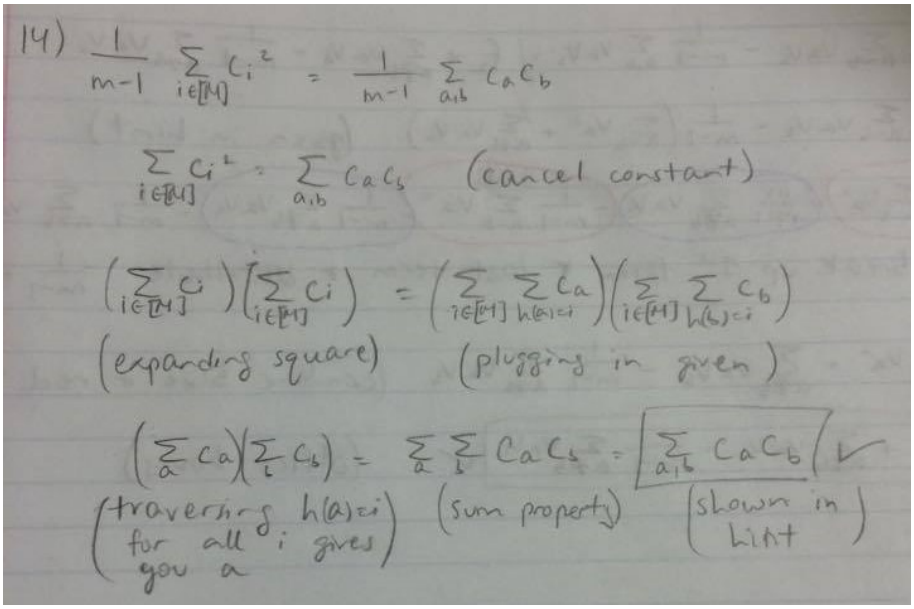
Problem 11: The empirical data line flattens out quickly because as d increases, we are checking more lines and picking the smallest of more lines. This increases our chances of getting a shorter line.

Second Moment Estimation

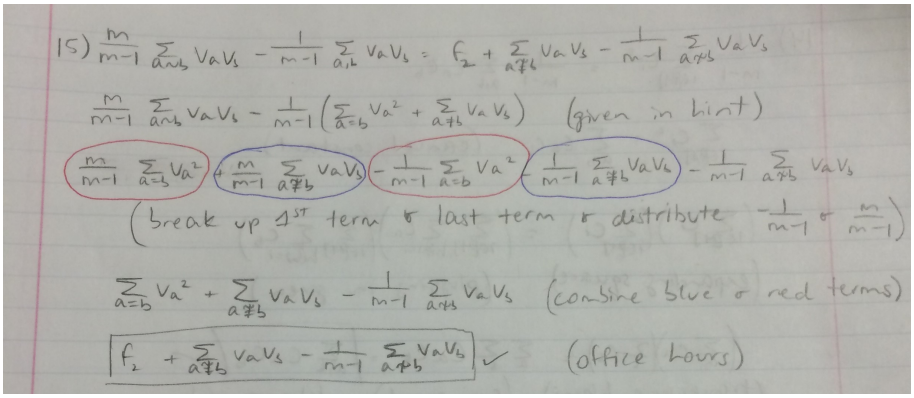
Problem 12: $v_d = 3$

Problem 13: F_2 of S = 18

Problem 14:



Problem 15:

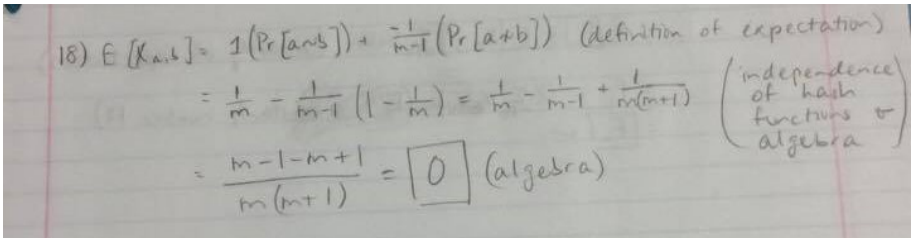


Problem 16: We can put $X_{a,b}$ where we did because it is conditioned on the two conditions found in the sums: either a and b are unequal but hash to the same bucket or they do not hash to the same bucket. The values based on the conditions are the same as the ones found in the equation that was proved in problem 15, so we can put $X_{a,b}$ in those places.

Problem 17: We can jump from line 1 to 2 because the two conditions in the sums combine to include all a and b that are not equal because if they are not equal there are 2 options: either they

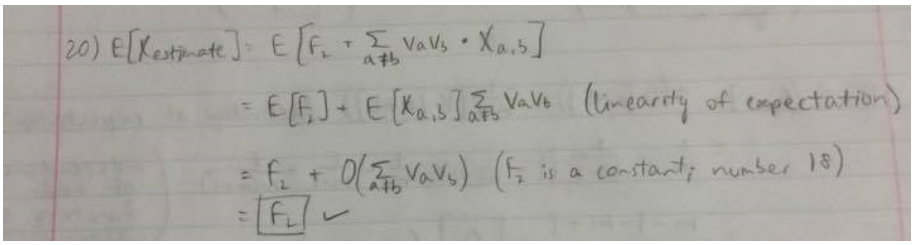
will hash to the same place or they will not.

Problem 18:



Problem 19: These two things are equal because $\sum_{a \neq b} v_a v_b$ can be treated as a constant and because of the linearity of expectation, you can pull it out of the expectation. (i.e. E[cX] = cE[X])

Problem 20:



Problem 21: We use the independence of hash functions when solving for $E[X_{a,b}]$ when finding the probability of two items hashing to the same bucket.

Problem 22:

```
def dataStream(self, numPackets):
        for i in range(numPackets):
            packet = self.packetGenerator()
            self.actualDataArray[packet] += 1.0
            hash_val = self.hash_4i(packet)
            self.counterArray[hash_val] += 1.0

    def actualF2(self):
        f2 = 0
        for x in self.actualDataArray:
            f2 += x**2
        return f2


    def estimateF2(self):
        est1 = 0
        est2 = 0
        for x in self.counterArray:
            est1 += x**2
            est2 += x
        est2 = est2**2
        est1 = est1*(self.m/(self.m - 1.0))
        est2 = est2*(1.0/(self.m - 1.0))
        return est1 - est2
```

```
    def __init__(self, m):
       self.m = m
       self.counterArray = [0]*m
       self.actualDataArray = [0]*(2**14)
```

Problem 23:

```
def F2_plot():
    x = []
    yf2 = []
    yest = []
    for m in range(50, 2501, 50):
        f2_est = SecondMomentEstimator(m)
        f2_est.dataStream(1000)
        yf2 += [f2_est.actualF2()]
        yest += [f2_est.estimateF2()]
        x += [m]
    plt.figure(1)
    plt.plot(x, yf2, color = 'r', linewidth = 3)
    plt.plot(x, yest, color = 'b', linewidth = 3)
    plt.title("Estimate vs Actual F2")
    plt.xlabel("m - Number of Buckets")
    plt.ylabel("Counts")
    plt.show()
```