Maoting Ren
mren

4-1. Expectation Versus “With High Probability” Results. Please do problems 23 and 28 from the math chapter in the textbook draft.

Solution:

23. a. p = 1/n , O(f(n) n)
(n-1)/n, O(f(n)/n)

In this example, it's expectation is E = 1/n * O(f(n) n) + (n-1)/n * O(f(n)/n) = O(f(n)) + O(f(n)/n) that is E = O(f(n)), but it's high probability is O(f(n)/n). So this example is O(f (n)) in expectation but not O(f (n)) with high probability.

b. p = 1/n , $O(f(n)\ n^2)$
(n-1)/n, O(f(n))

In this example, it expectation is E = 1/n *$O(f(n)\ n^2)$ + (n-1)/n*O(f(n)) = O(f(n) n) + O(f(n))
So the expectation is E = O(f(n) n), and high probability is O(f(n)).

28. assume we have a Las Vegas algorithm with expected running time O(f (n)), we need to run this algorithm (C log n) times to construct a Las Vegas algorithm whose running time is O(f (n) log n) with high probability.

4-2. Simulating a Biased Coin with Unbiased Coin Flips. Please do problem 26 from the math chapter in the textbook draft.

Solution:

we can do like this to simulating a biased coin with unbiased coin flips.
Firstly we flip one unbiased coin.

①**.** If the outcome is head, then we reduce the probability interval to [0, 0.5], if p is not in this interval, then we can output head. Else we flip the unbiased coin another time.

In the second flip, if the outcome is head, then we reduce the probability interval to [0, 0.25], if p is not in this interval, then we output head, if p is in this interval, we will flip another time again like before.

In the second flip, if the out come is tail, then we reduce the probability interval to [0.25, 0.5], if p is not in this interval, then we output tail, if p is in this interval, we will flip another time again like before

②. If the outcome is tail, then we reduce the probability interval to [0.5, 1], if p is not in this interval, then we can out tail. Else we flip the unbiased coin another time.

In the second flip, if the outcome is head, then we reduce the probability interval to [0.5, 0.75], if p is not in this interval, then we output head, if p is in this interval, we will flip another time again like before.

In the second flip, if the out come is tail, then we reduce the probability interval to [0.75, 1.0], if p is not in this interval, then we output tail, if p is in this interval, we will flip another time again like before

From the above algorithm, each time we have ½ probability to end the simulation, so we can expect only two flips for any probability p, that is in average we only need O(1) time to simulate a biased coin with unbiased coin

4-3. Average and Median Distance on a Line. Please do problem 66(b) from the sorting chapter of the textbook draft.

Solution:

Firstly, we sort the points by x. Then sweep all points we can get the smallest interval(call it

Min_Interval) and largest interval(call it **Max_Interval**) in O(n) time.
Then we can randomly select a interval value(call it **curInterval**) between [**Min_Interval**, **Max_Interval**]. As we have an interval value now, we sweep all points again and maintain a window which value is **curInterval** and a counter(call it **cnt_interval**) represent the number of intervals small than **curInterval** and **cnt_points** represent how many points in the window and **largest** represent the largest interval in the intervals smaller than **curInterval**. Then we use a pointer **left** to represent the left boundary of the window. Initially **left** is the first point, **cnt_interval** = 0**, cnt_points** = 1, **largest_k= 0**. Then we sweep from the second point. If the distance between current point and **left** is less than **curInterval**, then **cnt_interval** will increase by **cnt_points**, and **cnt_points** will increase by 1. Else if the distance between current point and **left** is large than **curInterval**, we will increase **left** until the distance is less than **curInterval,** then we update the value of **cnt_points** and **cnt_interval** and **largest_k**.

After we sweep all points, we can know how many intervals smaller than **curInterval** in O(n) time.
①If the value of **cnt_interval** is K, we are so lucky, the value of **largest_k** is just we want, that is kth largest of all pairwise distances.
②But if the value of **cnt_interval** is larger than k, it means there are more than k intervals smaller than **curInterval**, that is **curInterval** is a litter bigger than we want, so we randomly select a value from [**Min_Interval**, **curInterval**] as the new **curInterval**.
③If the value of **cnt_interval** is smaller than k, it means the number of intervals that smaller than **curInterval** is less than k, that is **curInterval** is a litter smaller than we want, so we randomly select a value from [**curInterval**, **Max_Interval**] as the new **curInterval**.

So in total, sort the points will cost O(n log n) time, and each time sweep all points, it will cost O(n). And random select will occur O(log n) times because each time it will divide the array into two half, and it will at most divide (log n) times. So the time complexity is O(n log n).

```
int findKthDistance(vector<int>& points)
{
    sort(points.begin(), points.end());
    int Min_Interval = 0, Max_Interval = points[points.size()-1];
    while(true){
        int curInterval = random(Min_Interval, Max_Interval);
        auto val = select(points, curInterval);
        if(val.first == k) return val.second;
        else if(val.first > k)
            Max_Interval = curInterval;
        else
            Min_Interval =   curInterval;
    }
}

pair<int, int> select(vector<int>& points, int curInterval)
{
    int cnt_interval = 0, largest_k = 0,  cnt_points = 1,  left = 0;
    for(int i =1; i < points.size(); i++)
    {
        if(points[i] - points[left] <= curInterval)
        {
```

```
                cnt_interval += cnt_points;
                cnt_points++;
                largest_k = max(largest_k, points[i] - points[left]);
            }
            else
            {
                while(points[i] – points[left] > curInterval)
                    left++;
            }
        }
        return  pair<int, int>(cnt_interval, largest_k);
}
```

4-4. Network Contention Resolution. Please do problem 25(b) in the textbook draft.

Solution:

1. In the first round, n/2 processors will attempt to transmit information, but they all fail due to the collision, so they will dormant.

2. In the second round, half of the remaining n/2 processors, that is n/4 processors, will attempt to transmit information, but they will fail too. And they will dormant too.

3. In the third round, half of the remaining n/4 processors, that is n/8 processors, will attempt to transmit information, but they all fail due to the collision, so they will dormant.

4. Each time, the processors which have not attempted to transmit information will reduce by half, until there is only one processor left, and he will successfully transmit his information. And next time, no processor will transmit information, so the dormant processors will wake up, and do the same thing like first round.

And we can see from the above analyze, each time the processors which have not attempted to transmit information will reduce by half, and we can expect that it will cost O(log n) time steps to elapse between successful transmissions.