CSCI 5454
Alimulla Shaik
No Collaborators

1) a) Sol.
We are given a subroutine probefluxcapacitor() which outputs each time an independent bit, 1 with probability p and 0 with probability 1 □- p. But probability p not necessarily equal to ½.

Algorithm for randbit():
Steps: 1. Consider an hash table with two entries, namely Hash (1) and Hash(0) which contains bit value and its frequencies. Initialize both frequncies to ZERO. 2. Call probefluxcapacitor() twice to get random bits in pairs (x, y) with probability . 3. probefluxcapacitor() = (x, p) 4. probefluxcapacitor() = (y, 1 - p). 5. If x = 1, then update or increment its corresponding hash bit frequency. i.e, Hash (1) otherwise, update or increment frequency value in Hash (0). 6. If y =1, then increment Hash(1) else increment Hash (0). 7. If frequencies of both hash keys are same then return x,otherwise continue step 2.

Correctness:
This algorithm reads input in pairs using given subroutine. We are updating hash key values each time based on the bit key. After reading both pair values, we will check frequencies of both hash keys and if both values are same then return corresponding bit. This frequencies equality ensures that both random bits are having probability exactly ½ and probability of both bits to occur is ½ which means they are uniformly random. Consider an example, let's say first time calling subroutines outputs (1, 1) then Hash (1) = 2 and Hash (0) = 0. Both key values are not same so we will continue the process. Let's say this time subroutine returns (0, 0) and Hash (1) = 2 and Hash (0) = 2, here both key values are same so algorithm returns. This ensures that probability of both bits to occur is exactly ½ and thus can say that they are uniformly random.

Running Time:
Maintaining hash table with 2 entries and to update both key values (frequencies) each time takes O(1) time. Call to subroutine probefluxcapacitor takes constant time as it returns either 1 or 0 with probability p or 1 - p. We are reading input in pairs here, so the possibilities of calling subroutine gives us either (1, 0) or (0, 1). After considering probabilities for the same, we will get (p, 1 - p) or (1 - p, p).
The probability of this algorithm is (p)(1 - p) + (1 - p)(p) = (1 - p)(p + p) = (1 - p)(2p)
Hence, Running time of this algorithm in terms of probability p (after ignoring constant times) is O(p (1-p)).

1) b) Sol.

Algorithm for randbit(p)
Steps: 1. Consider randbit() from the above and bit variable to store all outcomes of randbit function in binary form. 2. For each i in n, call randbit() and keep an adding or append output value to bit variable. 3. For all the possibilities, if value of bit variable which is in binary form, is less than for some integer 'k' then return 1, else return 0.

Correctness:
We need to able to represent randbit(p) in terms of $p = k / 2^n$ for some integers k, n. We need to call randbit() function n times which generates different possibilities. To understand the algorithm let's consider an example with n = 3 and k = 3.
randbit(p) ⇒ calls randbit() 3 times and generates 8 different possibilities. They are,
{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0),(1, 1, 1)}
Here k = 3 so check for the possibility which has < k.
⇒ value of binary form (0, 0, 0) = 0, (0, 0, 1) = 1, (0, 1, 0) = 2
⇒ So for all the possibilities, only 3 values are < k.
The probability of p for this algorithm , Pr[randbit(p) = 1] = p = 3/8
Which can be expressed as $p = 3 / 2^3 = k / 2^n$
Now, we can say that $Pr[randbit(k / 2^n) = 1] = k / 2^n$

1) c) Sol.
Algorithm:

```
def randbit(p):
        for i = 1,2,3,...:
        let d = getdigit(p,i)
        if randbit() != d, return d
```

Correctness:
We are given that getdigit(p,i) returns ith digit in the binary representation of p. In the algorithm, we are comparing ith digit with output of randbit(). If both are not equal then algorithm returns ith digit. Otherwise, it continues the loop. If ith digit equals to randbit() outcome then it means that, probability 'p' is formed with bits generated by randbit and can ignore those values. This algorithm ensures that, it will output different random bits which are not part of probability. So this will work for any probability and it need not be in the form of $k / 2^n$ for any k, n integer values.

Running Time:
We are given that, getdigit(p,i) runs in polynomial time, in $O(n^c)$ for some c.
We are running the loop to compare ith digit with randbit function, n times so its complexity is O(n). We can compare values of both functions getdigit and randbit in constant time which we can ignore. Hence, running time of this algorithm is $O(n * n^c) = O(n^{c+1})$ for some c.

2) Sol.

Algorithm:
Steps: 1. Let’s assume that given array as arr[n]. 2. Consider variables a and count and make a = 0, count = 0; 3. If n ==1 ⇒ return YES as it is the only element in the array. 4. For each element i in array arr,   if count == 0 ⇒ a = i;   if arethesame(a, i) ⇒ count = count + 1;   else count = count - 1; 5. If count > 0 ⇒ return YES   else return NO

Correctness:
This algorithm checks for elements appears more times in array and specifically n/2 times. My approach is to store initial element in a variable ‘a’ and start traverse elements of the array. Use the given function arethesame(i, j) to know that elements are same or not. If next element in the array is same as ‘a’ then increment the count, decrease otherwise. If any element which appears more than n/2 times in array exists then count for this approach must be positive integer (> 0). So we are verifying count value at the end of algorithm to ensure it is > 0. If it is then array has such element and return YES, else returns NO. If there is only one element present in the array then simply return YES as it appeared more than ½ times.

Example:
Given Array arr[]: 1 2 1 3 1 1
Traverse each element in arr, for each element in arr:
⇒ First iteration, initially count = 0 ⇒ a = 1
⇒ Next iteration i = 1, arethesame(0, 1) = NO ⇒ count = -1;
⇒ i = 2, arethesame(0, 2) = YES ⇒ count = 0;
⇒ i = 3, arethesame(0, 3) = NO ⇒ count = -1;
⇒ i = 4, arethesame(0, 4) = YES ⇒ count = 0;
⇒ i = 5, arethesame(0, 5) = YES ⇒ count = 1;
At the end of the algorithm, count > 0 ⇒ So there exist an element which appeared more than n/2 times and this algorithm will return YES.

Running Time:
This algorithm traverses all the elements of the array and it takes O(n) time. We can keep track of counter with comparisons using arethesame() in constant time, O(1). Hence, running time of this algorithm is O(n).

3) a) Sol.
We are given that minion is added to random pod every second. And the minions are distributed among ‘k’ pods. After ‘t’ seconds, we can determine number of minions by

combining minions in ‘k’ pods. We need to calculate expected number of minions in the first pod only and let’s consider expectation formula.

From the definition of expectation, for any random variable X and event E,

$E(X) = \sum_{x=0}^{\infty} X(a)\, P(a)$

Here, consider random variable X as number of minions in the first pod. To find the probability of the same, we can consider, binomial distribution [1].

The probability of getting exactly k successes in n trials is $\Pr[X = k] = {}^{n}C_{k}\, p^{k} (1 - p)^{n-k}$

So, the probability of number of minions ‘n’ in the first pod after t seconds

$\Rightarrow \Pr[X = t] = {}^{n}C_{t}\, (1/k)^{t} (1 - (1/k))^{n-t}$

Hence, expected number of minions in the first pod after ‘t’ seconds

$E(X) = \sum_{x=1}^{t} x\; {}^{n}C_{t}\, (1/k)^{t} (1 - (1/k))^{n-t}$

This is in the same form as $\sum_{i=1}^{n} x_i\, p_i = np$ So, $E(x) = x((1/k))$

3) b) Sol.

We are given that every second, one minion gets added to some random pod. To calculate when will the first pod gets its first minion, we need to check how many minions were added in rest of the pods before first pod got its minion. We can determine expected number of minions were added to rest of the pods before first pod got its using geometric distribution [2].

If the probability of success on each trial is *p*, then the probability that the k^th^ trial is the first success is

$\Pr[X = k] = p\, (1 - p)^{k-1}$

So required expectation $E(X) = \sum_{x=1}^{\infty} x \;\; (1/k)\, (1 - (1/k))^{x-1} = k$.

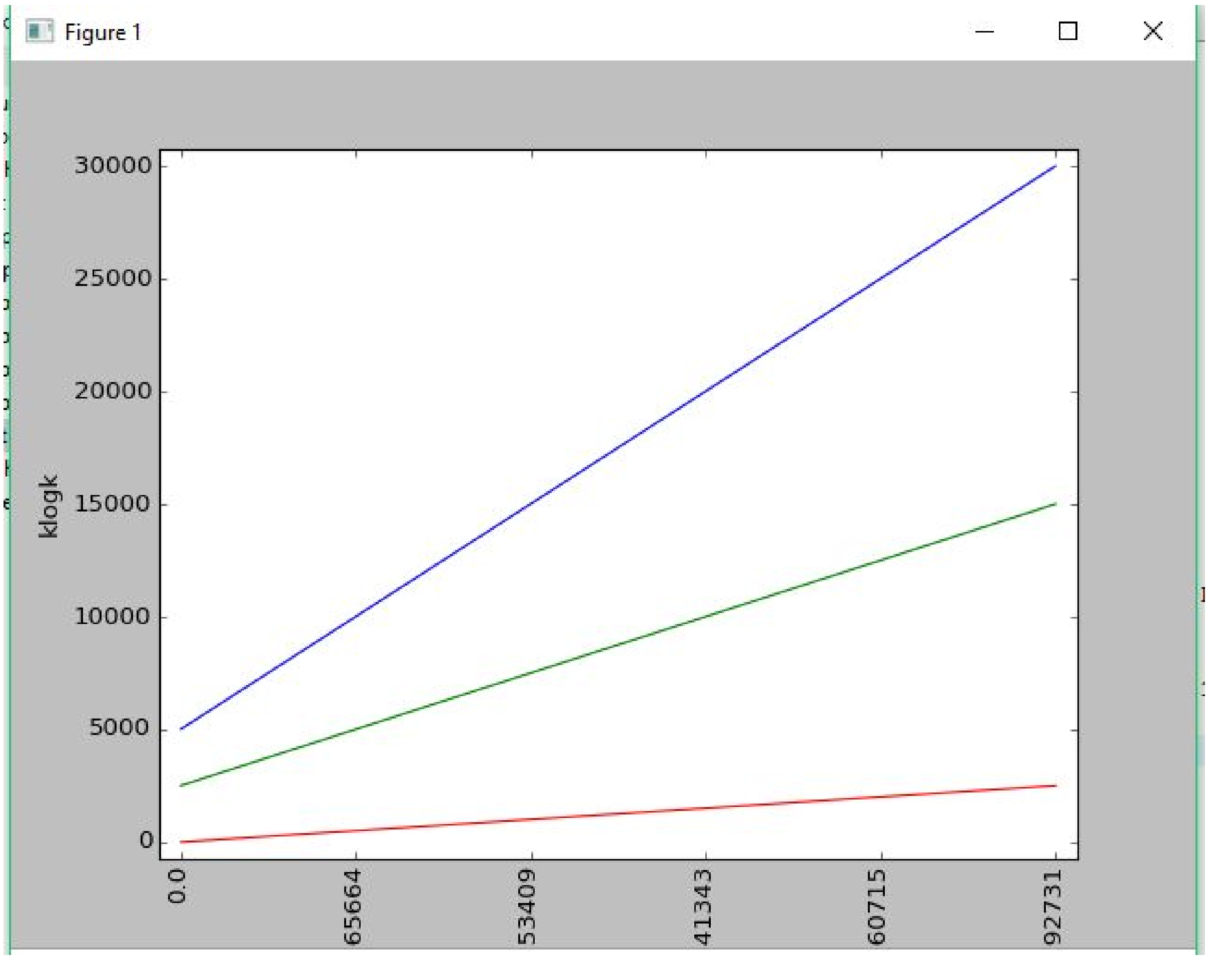
3) c) Sol.

To determine, when will all pods have at least one minion, first we will find out the probability that the i^th^ pod will be filled is (k □- n + 1)/k. The expected time until the i^th^ pod is filled with at least one minion is k/(k □- n+1). So the expected value is $E[X] = \sum_{n=1}^{k}$ k/(k □- n+1)

$E[X] <= k \sum_{n=1}^{k} 1/n$

3) d) Sol.

Required graph is shown below.



4) a) Sol.
Karger’s algorithm chooses an edge randomly and merges two vertices of the edge to make it one vertex. Then reconnects of the edges. As we made two vertex into one, edges involved with these two vertices will be contracted. Eliminate self loops if there are any. Continue these process by merging vertices and then reconnecting edges until two vertices remain. On the other hand, spanning tree is a subgraph that includes all the vertices of the graph. We can say that, Karger’s algorithm returns a subgraph with only two vertices of the given graph, which is similar to spanning tree definition of the graph.

4) b) Sol.
As explained in above question, Karger’s algorithm chooses an edge randomly but kruskal algorithm chooses an minimum weight edge in uniformly distributed random weights. Both algorithm chooses their first uniformly random edge with same probability distribution. To elaborate the same point, let’s consider if we have ‘n’ edges then selecting a random edge using karger algorithm in uniform probability distribution is 1 / n which is same as selecting an minimum weight edge in kruskal. For the first edge selection both algorithms have same probability.
For the next edge Selection, karger selects uniform random edge again whereas kruskal selects minimum weight edge of remaining edges. Clearly, both selects an edge with same probability again.

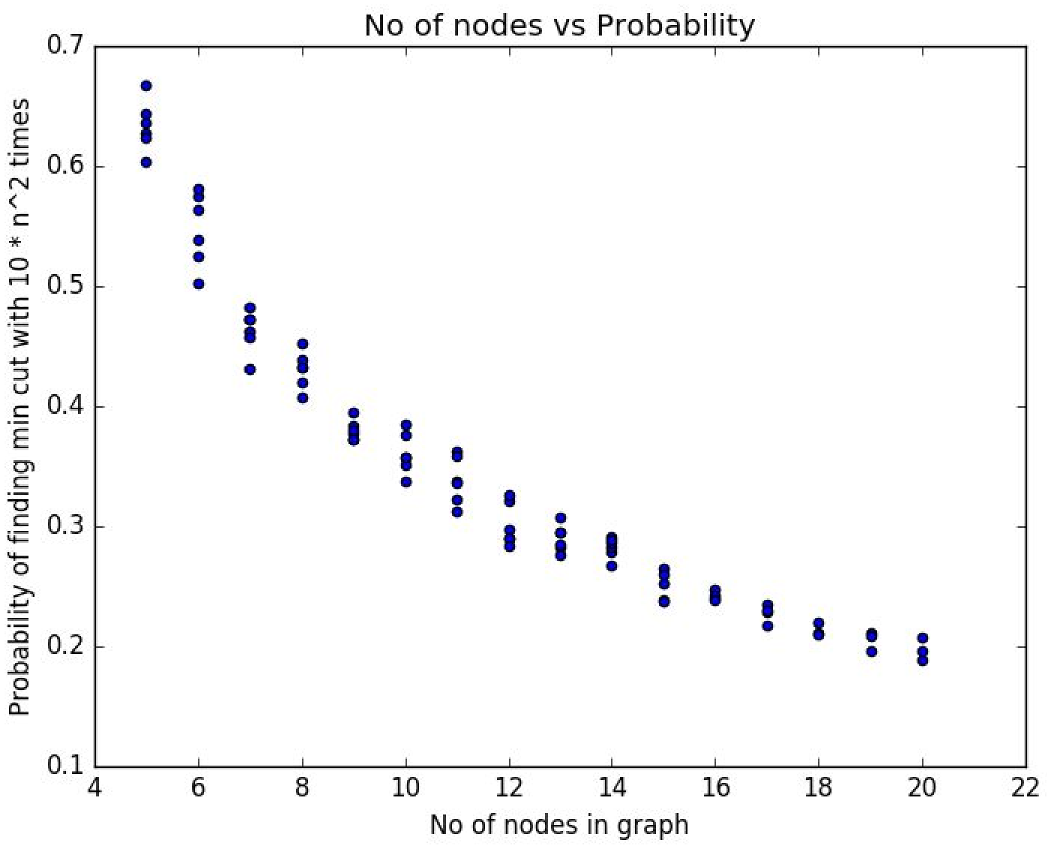
While moving on using karger algorithm, some choices will result in different outcomes than what we were actually expecting but still it does not affect anything because we are always selecting an uniform edge in both algorithms. We are constructing spanning trees with help of the edges and it's selection matters in both the algorithms which has same probabilities. So the probability of selecting spanning tree of T in both algorithm is same.

4) c) Sol
As we know that, kruskal starts with minimum weight edge and then continuously selects next minimum and so on. At the end of the algorithm, edge remaining will be having maximum weight. In karger, we starts with random edge and then merge two vertices into one until entire graph has turned into two vertices. Final cut in it surely involves last remaining edge after all merging and reconnection of edges. So the final cut output by karger is same edge as maximum weight edge in kruskal algorithm.

4) d) Sol.
From the a) part, we proved that karger algorithms produces spanning tree like kruskal algorithm. In part b) we proved that both algorithms chooses an edge with same probabilities. In part c) we proved that final cut output by karger is same edge as maximum weight edge in kruskal algorithm. After combining all these 3 parts, we can say that they both return same cuts of vertices.

4) e) Sol.
It's easier to run karger's algorithm with one run takes $O(n^2)$ time.
Therefore, this randomized algorithm takes $O(n^4 \log n)$ time.
To optimize this algorithm further, Karger and Stein came up with an idea to run algorithm for $(1 + n / \sqrt{2})$ vertices and to use recurse on resulting path.
Now, The probability to find the find a specific cutset 'K' is given by the recurrence relation
$T(n) = 2 (n^2 + T(n / \sqrt{2})) = O(n^2 \log n)$

5) Sol.
The graph with given data is shown below.



6) Sol.
We are given that players are lined up in a long narrow corridor with a door in the front and the back. And captains can only select the player who are currently either in the front or the back of the line. We are given that every player has some abilities x_i >0. To derive an optimal strategy to select the players, maximizing the sum of player abilities, first we will understand different possibilities each captain's has.

My strategy is to consider an element and try to maximize capabilities for (n - 1) and minimize the options for B to select player with maximum capabilities.
Assume player capabilities like below.

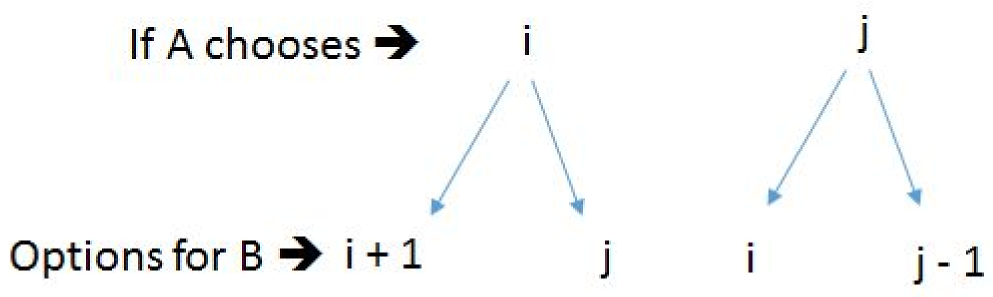
4	15	30	10

If A has picked 10 then it will give choice to B to pick up 30 which is the player with most capabilities factor. So in these kind of situations, instead of going for higher capabilities, will choose to pick '4' which gives B either 15 or 10 but not highest capabilities. Once, B has made choice, A can select 30.

We can always convert this problem into sub- problems as shown in below.

i	i+1				j - 1	j

⇑ ⇑



As you can see If A selects an option then B has two choices as mentioned in above diagram. Again, each leaf nodes in above diagram {(i +1), j, i , (j -1)} will have similar choices like above with different positions. This will continue till we reaches the end of the line up. We will apply the strategy for sub problems and will make use of the same to determine strategy for the entire problem. Clearly this problem is in DP (Dynamic Programming).

Algorithm:
Steps: 1. Consider an array with 'n' elements to represent line and front will be at Arr[0] and back end will be at Arr[n - 1]. Consider two captains A and B where A chooses first. 2. As mentioned above, each captain has two choices and based on first captain choice, next captain will have another two choices to select. 3. My strategy is to maximize capabilities for remaining elements (n - 1) and minimize the options for B to select player with maximum capabilities. 4. For A, options to select players is option = Max { (Arr[i] + min(Arr[i+1], Arr[j])), (Arr[j] + min(Arr[i], Arr[j -1])) } 5. Once, A is selected, now its B turn to select. 6. B can follow any algorithm of its choice and once it made selection, follow the step 4 with corresponding available positions of the players. 7. If there are only two players left then choose player with maximum of capabilities. 8. Continue the step 4 till all players are selected.

Correctness:
This strategy works fine for to maximize players capabilities. We applied optimal strategy using
Dynamic Programming. We are maximizing player capabilities each time and minimize B chances to select at the same time.

Running Time:
If the players size is n then this algorithm has a runtime of O(n). It will loop through all the players until there are only two left.

7) a) We are given that, all n minions designated a unique location. If first minion did not find its position then it chooses some random position and rest all minions also selects some random position if their designated spot is taken else they will pick their designated spot.

Now, the probability of first minion to select its position = 1/ n as it can select any available n positions. For any minion ‘i’, if its designated position is empty then it has to take that spot and therefore can not occupy last position.
So the probability for a minion ‘i’ to occupy the it’s position (not last position) is P(i) = P(i - 1)
If for minion ‘i’, its designated position is already taken then it has to pick a random spot. At this point, i -1 positions are already filled, so minion can take any (n - i - 1) positions available and probability for the same is P(i) = 1 / (n - i - 1).
We can use this relation to calculate probability of first minion to pick a spot and it gives P(1) = 1 / n
Now we have determined the probability that last position is not taken in all the cases.
Means, last minion will occupy that last position which can lead Gru to launch.
Hence, The recurrence relation for Gru can launch is P(i) = P(i - 1) + 1 / (n - i - 1)

7) b) First, check whether the last position is filled or empty. If that position is already taken then check which minion occupied that spot. Track last position occupied minion’s original position and filled it in his designated position. Now find the position where current left out minion’s position and designate correspondingly. Repeat this from backwards to re-arrange minions so that no minion is left out except last minion. This re-position will ensure that last position is empty and we will fill that by last minion which has launch codes. So now Gru can launch without any problems.

Reference:
1. https://en.wikipedia.org/wiki/Binomial_distribution
2. https://en.wikipedia.org/wiki/Geometric_distribution
3. https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
4. https://en.wikipedia.org/wiki/Karger%27s_algorithm