Rutam Kathale September 29, 2023

CS-1203 Problem Set 1

Collaborators: *none*

Problem 1-1. Counting and Sets

Suppose the letters a, b, c, d have proper positions 1,2,3,4 i.e. the correct sequence is abcd. Write down all the deranged sequences (where none of the letters are in their proper position). Find a generalised combinatorial expression for n letters and use it to verify your answer.

Solution: We can initially find the derangements for a,b,c,d as:
badc, bcda, bdac, cadb, cdab, cdba, dabc, dcab, and dcba.
As a result, we got 9 derangements for a,b,c,d. While calculating derangements manually can be convenient until n (number of variables/symbols), becomes really large. For that, we have the derangement formula which gives us the number of derangements for 'n'.
number of derangements =

$$n!\left(1-\frac{1}{1!}+\frac{1}{2!}-\frac{1}{3!}+\ldots+(-1)^n\frac{1}{n!}\right)$$

Using the formula for n = 4 above, we get
24(1- 1 + 1/2 - 1/6 + 1/24) = 24(1/2 - 1/6 + 1/24) = 12-4 +1 = 9

So, hereby we have verified the number of derangements on a,b,c,d from the formula.

Problem 1-2. Probability Practice: Dice Rolling

Asami and Bolin are playing a dice game in which a pair of dice is rolled repeatedly. Asami wins if a 6 is rolled before any number greater than or equal to 9, and Bolin wins if any number greater than or equal to 9 is rolled first. We will find the probability that Asami wins the game

(c) Compute

$$\sum_{n=1}^{\infty} P(E_n)$$

and argue rigorously that it is the desired probability.

Solution:
The ways to get a sum of 6 are: (1,5), (2,4), (3,3), (4,2), (5,1). So there are 5 ways to get a sum of 6. Therefore, the probability of getting a sum of 6, P(6), is 5/36. The ways to get a sum greater than or equal to 9 are: (3,6), (4,5), (4,6), (5,4), (5,5), (5,6), (6,3), (6,4), (6,5), (6,6). So there are 10 ways to get a sum greater than or equal to 9. Therefore, the probability of getting a sum greater than or equal to 9 is 10/36 = 5/18.

We want to find the probability that a 6 occurs on the nth roll and neither a 6 nor any number greater than or equal to 9 occurs on any of the first n-1 rolls.
The probability that neither a 6 nor any number greater than or equal to 9 occurs on a single roll is P(l) = 1 - P(6) - P($\geq$ 9) = 1 - 5/36 - 5/18 = 21/36 = 7/12.
Therefore, the probability of E_n is P(E_n) = $P(neither 6 or \geq 9)^{(n-1)}$ $*$ P(6) = $(7/12)^{(n-1)}$ $*$ (5/36).
The desired probability is

$$\sum_{n=1}^{\infty} P(E_n)$$

.
It is a geometric series with a = P(E_1) = P(6) = 5/36 and common ratio r = P($neither 6 nor \geq$ 9) = 7/12.
The sum of an infinite geometric series S = a / (1 - r).
So, S = P(E_1) / (1 - P($neither 6 nor \geq$ 9)) = (5/36) / (1 - 7/12) = 15/31. So the probability that Asami wins the game is 15/31.

(d)
1. Given that each of these ten rolls results in neither a 6 nor any number greater than or equal to 9, enumerate the possible totals for each roll and their respective probabilities.
Solution: The respective probabilities (sum of the pair of dice) are as follows:
2: ((1,1)): P(2) = 1/36

3: ((1,2), (2,1)): P(3) = 2/36 = 1/18
4: ((1,3), (2,2), (3,1)): P(4) = 3/36 = 1/12
5: ((1,4), (2,3), (3,2), (4,1)): P(5) = 4/36 = 1/9
7: ((1,6), (2,5), (3,4), (4,3), (5,2), (6,1)): P(7) = 6/36 = 1/6
8: ((2,6), (3,5), (4,4), (5,3), (6,2)): P(8) = 5/36
We can't have 6 or number greater than 9

2. For these ten rolls, let X_i be the number on the face of the die for roll i. Compute the expected value of X_i.
Solution: We can use the above probabilities to get the expected value
E[X_i] = (2 * (1/36)) + (3 * (1/18)) + (4 * (1/12)) + (5 * (1/9)) + (7 * (1/6)) + (8 * (5/36)) $\approx$ 5.833. So the expected value of X_i is approximately 5.833.

3. Compute E[X] where X is the sum of these ten dice rolls.
Solution: Expectation is linear and so, E[X] = E[$10 * X_i$] = 10 E[X_i] = 58.33

4. Compute the variance of X, where X is again the sum of these ten dice rolls.
Solution:
Var(X_i) = E[X_i^2] - $(E[X_i])^2$
We have E[X_i] = 5.833. So, E[X_i^2] = [$xi^2 *$ P(xi)] for all i

E[X_i^2] = ($2^2 *$ (1/36)) + ($3^2 *$ (1/18)) + ($4^2 *$ (1/12)) + ($5^2 *$ (1/9)) + ($7^2 *$ (1/6)) + ($8^2 *$ (5/36)) $\approx$ 38.611.
Var(X_i) = E[X_i^2] - $(E[X_i])^2$ = 38.611 - $(5.833)^2$ $\approx$ 6.3.
$Var(X)$ = 10*Var(X$_i$) = 10 $*$ 6.3 $\approx$ 63.

5. Use Chebyshev's inequality to find an upper bound for P[$\|X1220/21\| \geq 10$].
Solution: According to Chebyshev's inequality, the probability that the variable deviates from its mean by more than k standard deviations is at most $1/k^2$.
We have E[X] is 58.33. The deviation from the mean is 10 (from the expression in question), and the standard deviation is the square root of the variance, which is $\sqrt{(}63)\approx$7.99.
So we can calculate k as the deviation divided by the standard deviation: k = 10 / 7.99 $\approx$ 1.25.
We get:

P[$\|XminusE[X]\|$ $\geq$ kσ]$\leq$ $1/k^2$
P[$\|Xminus58.33\|$ $\geq$ 10] $\leq$ $1/(1.25)^2$
P[$\|Xminus58.33\|$ $\geq$ 10] $\leq$ 0.64 So, the probability that the sum of these ten dice rolls deviates from its expected value by 10 or more is at most 0.64 giving us our upper bound.

Problem 1-3. Su's Symmetry Studies

(c) Suppose the planar deposition is divided into a grid of squares, where each square has dimensions 1/2 × 1/2 . Why must you reject G if two atoms are in, or on the boundary of, the same square?
Solution:
Given that the planar deposition is divided into grid of squares, we can denote each square in the grid by list of atoms that fall within it.
If there are two points g_i and g_j in the same/neighboring squares in the grid, their Euclidean distance cannot be greater than 1 because the diagonal distance within a $1/2 \times 1/2$ square is $\sqrt{2}/2 \approx 0.71$.
If two atoms are in, or on the boundary of, the same square, then the maximum possible distance between them would be the diagonal of the square. So, if two atoms are in or on the boundary of the same square, by using the $1/2 \times 1/2$ squares, their distance is less than 1 angstrom, and therefore, it violates the property that each atom must be at least 1 angstrom away from every other atom. This is why G must be rejected in this scenario.

(d) Propose an algorithm to determine whether there exists a pair of atoms less than 1 angstrom apart, providing a proof of correctness and a runtime analysis. For full credit, your algorithm should run in O(n log n) time with respect to n = $\|G\|$, the number of atoms.
Hint: Use divide-and-conquer with part (c).
Solution:
Similar to merge sort, we can implement divide and conquer algorithm to the problem in 2 stages - divide stage and merge stage.

Divide Stage:
1. Sort the atoms in G on their x-coordinates in $O(n \log n)$ time.
2. Divide the sorted list of atoms into two equal halves based on the median, creating two sublists (similar to quicksort where we arrange the pivot element such that all elements in left part are smaller and those in right are larger): G_l and G_r.
3. Loop through (via recursion) G_l and G_r to find the closest pairs. Let the minimum distances in the left and right sublists be d_l and d_r, respectively.
4. Take the minimum d of d_l and d_r.

Merge Stage:
1. Find the median x-coordinate x_m
2. Gather the atoms within a distance d of x_m in the sorted list in $O(n)$ time.
3. Sort it on their y-coordinates in $O(n \log n)$ time.
4. Repeat
Then we return the minimum among d and the smallest distance found.

Proof of Correctness:
The divide stage identifies the closest pairs in the left and right sublists, and the merge step considers pairs that cross the boundary between the two sublists.
If there exists a pair of atoms in G with a Euclidean distance less than 1 angstrom, they must be in left sublist, the right sublist, or they can be between the two sublists.
By recursively finding the closest pairs in the left and right sublists and considering the part of atoms around the median (similar to merge sort where we consider left part, right part, and part comprised of left and right extremes), the algorithm correctly identifies the closest pair in the entire set G.

Runtime Analysis:
Sorting the atoms by x-coordinate and y-coordinate each takes $O(n \log n)$ time. The divide stage takes $O(n \log n)$ time, and the merge step takes $O(n \log n)$ time (since we sort the atoms within distance d of x_m).
So, the overall complexity of algorithm would be O(nlogn)

(e) Give a solution to determine whether there exists a pair of atoms a_i, a_j A, i $\neq$ j such that the distance between the two atoms in the 3D space is less than 1 angstrom. If there is such a pair, your algorithm should return an example pair, and if there is no pair, your algorithm should return false. Prove the correctness of your algorithm, and analyze its runtime. For full credit, your algorithm should run in O(n log^2 n) time with respect to n = $\|A\|$, the number of atoms.
Solution: We can, again use the divide-and-conquer algorithm used in the 2D setting. The algorithm would as previously, have two stages - divide stage, and merge stage.
Divide Stage:
1. Sort the atoms in A based on their x-coordinates in $O(n \log n)$ time.
2. Divide the sorted list of atoms into two equal halves by the median x-coordinate, creating two sublists: A_l and A_r.
3. Recursively apply the algorithm to A_l and A_r to find the closest pairs within each sublist. Let the minimum distances in the left and right sublists be d_l and d_r, respectively.
4. Take the minimum of d_l and d_r and call it d.

Merge Step: 1. Find the median x-coordinate x_m of the atoms in the sorted list. 2. Gather atoms within distance 'd' of x_m in sorted list (O(n)).
3. Sort it based on their y-coordinates in $O(n \log n)$ time.
4. Iterate to find pairs of atoms with a Euclidean distance less than d in 2D. This is a step where the 3D implementation differs from 2D. Here, for each pair (a_i, a_j) in 2D, we also have

to additionally calculate their 3D Euclidean distance. This can be done by taking square root of squares of difference of x_i and x_j, y_i and y_j, and z_i and z_j.
5. If we get any distance less than 1 angstrom, return that pair. Otherwise, we just return false if no pair with a 3D distance less than 1 angstrom was found

Correctness Proof:
The divide stage finds the closest pairs in the left and right sublists.
The merge step considers pairs that cross the boundary between the left and right sublists and checks their 3D distances. If there exists a pair with a 3D distance less than 1 angstrom, it will be returned. So, the algorithm would correctly identify the closest pairs, by dividing and returns pairs with 3D distance less than 1 angstrom.

Runtime Analysis:
Sorting the atoms by x-coordinate and y-coordinate each takes $O(n \log n)$ time.
Divide step takes $O(n \log^2 n)$ time as we are dividing and also sorting the list.
Then, we proceed to merge step which involves sorting based on y-coordinates in $O(n \log n)$ time and then iterating through it to check distances in 2D and calculate distances in 3D. The overall runtime of the merge step is $O(n \log^2 n)$.
So, the overall runtime of the algorithm is $O(n \log^2 n)$.

Problem 1-4. Land It or Not

(a) You need to design a system that efficiently handles landing requests and keeps a record of approved requests. Can you use arrays to keep track of this? If so, briefly explain how you would do so.
Solution:
We can use arrays to handle landing requests and keep a record of approved requests. We can use the below algorithm:
1. When a new landing time is requested, iterate over the array (of pre-approved requests) and check if the absolute difference between the requested time and all approved times is greater than the buffer time 'k'. If it is, approve the request; otherwise, decline it. We keep the array of pre-approved elements sorted for processing new requests faster.
2. If the request is approved, find the correct position in the array to insert the new time so that the array remains sorted. This can be done using binary search.
3. Since we're keeping our array sorted, we can simply print out the array to get a sorted order of landings.

This approach has a time complexity of O(n) for each insertion due to shifting elements when maintaining sorted order in an array apart from initially sorting the array in O(n log n) using quick sort/ merge sort.

(b) Explain how you could also use binary search trees for this. Compare the efficiency of the BST solution and the array solution.
Solution: We can use BST for managing the requests as:
1. When an insertion of a spot is requested, we can search the BST to find the predecessor and successor. If the new time minus the predecessor is greater than 'k' and the successor minus the new time is also greater than 'k', then approve the request; otherwise, decline it. If the request is approved, we insert it. 3. We can perform an in-order traversal of the BST, to print out the landing times.

For each request, both solutions need to search for where to insert a new time. The array solution uses binary search, which is O(log n), while searching in a BST is also O(log n). For an array, after finding where to insert a new element, we need to move all elements on right taking O(n) time. For a BST, once we've found where to insert a new node, we can do so in O(1) time (and finding the location to insert would take O(n logn)).

Problem 1-5. Enchanted Forest

Explain how you would keep track of the C-th smallest node at any given time as numbers are being added to the Tree of Wonders. Make sure you explain any data structures you would use, and how they help you to solve the problem efficiently.
Solution:
We can use a Max Heap to keep track of the smallest C elements.
The max heap will store the smallest C elements encountered so far.
When a new number comes in, there are two cases:
1. If the heap has less than C elements, we insert the number into the heap.
2. If the heap has C elements and the new number is smaller than the maximum element in the heap (root), replace the maximum element with the number.
3. After each insertion or replacement, we heapify the heap.

So, the root of the Max Heap is always the C^{th} smallest element encountered so far.
Also, If a new number is greater than the root of the Max Heap (which is the C^{th} smallest element so far), we don't care because we're only interested in the smallest C elements. If it's larger than all C elements currently in our Max Heap, so it can't possibly be the C^{th} smallest element.
Both insertion and deletion in a binary heap take O(log n) time, where 'n' is the size of the heap. Therefore, processing each incoming number takes O(log n) time.