COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 3 Solutions

Announced on: Aug 08 (Thurs)

Problems marked with ($\star$) will not be asked in the tutorial quiz.

1. In this problem, we will design an algorithm to fairly divide a (mathematical) cake among a bunch of kids. Let the interval $[0, 1]$ denote a *cake*. There are n kids, and kid i's preferences over the cake are given by a nonnegative and integrable *value density* function $v_i : [0, 1] \to \mathbb{R}_{\geqslant 0}$, which should remind you of the probability density function. For any subinterval (or *piece*) $[a, b] \subseteq [0, 1]$ of the cake, kid i's value for it is given by $V_i([a, b]) := \int_a^b v_i(x)dx$. We will assume that every kid values the entire cake at 1, i.e., for every i, $V_i([0, 1]) := \int_0^1 v_i(x)dx = 1$.

   A cake *division* $X = (x_1, x_2, \ldots, x_n)$ refers to a partition of the cake into n subintervals $[x_0, x_1]$, $[x_1, x_2]$, $\ldots$, $[x_{n-1}, x_n]$ where $x_0 = 0$ and $x_n = 1$.[1] A division is said to be *proportional* if each agent's value for its piece is at least $1/n$ of its value for the entire cake. For example, if $n = 3$ and kid 1 gets the piece $[0, x_1]$, kid 2 gets $[x_1, x_2]$, and kid 3 gets $[x_2, 1]$, then proportionality would require that $V_1[0, x_1] \geqslant 1/3$, $V_2[x_1, x_2] \geqslant 1/3$, and $V_3[x_2, 1] \geqslant 1/3$.

   The valuations can be accessed via two types of *queries*:

   - a $\texttt{cut}(i, x, \alpha)$ query, which, given an index i, a starting point $x \in [0, 1]$ on the cake and a value α, returns a point $y \in [0, 1]$ on the cake such that kid i's value for the piece $[x, y]$ equals α (if such a point exists) and return `null` otherwise, and
   - an $\texttt{eval}(i, x, y)$ query, which, given an index i and two points $x, y \in [0, 1]$ on the cake such that $x \leqslant y$, returns kid i's value for the piece $[x, y]$.

   a) Suppose there are $n = 2$ kids. Design an algorithm that always computes a proportional cake division using at most two queries.

   b) Let us now talk about a general number n of agents. Consider the following algorithm: Each agent i makes a mark at point x_i such that its value for the piece $[0, x_i]$ is exactly $1/n$. The agent with the leftmost mark is given the enclosed piece, and the algorithm resumes with the remaining agents and the remaining cake. Is this algorithm proportional? How many queries does it need in the worst case?

   c) For a general n, design a divide-and-conquer algorithm that, given any preferences of the kids, always returns a proportional cake division using at most $\mathcal{O}(n \log n)$ queries. Is this algorithm faster or slower than the one in part (b) (in terms of number of queries)? You may assume n to be a power of 2.

   d) For part (c), can you design an algorithm without using $\texttt{eval}(\cdot)$ queries?

[1] The endpoints of the subintervals have zero value and can be arbitrarily assigned to either of the adjacent pieces.

a) **Solution 1**: We can solve the problem using 2 cut queries as follows:

i. $x_1 = \texttt{cut}(1, 0, 1/2)$

ii. $x_2 = \texttt{cut}(2, 0, 1/2)$

The first query finds the point $x_1 \in [0, 1]$ such that $V_1[0, x_1] = 1/2$. The second query finds the point $x_2 \in [0, 1]$ such that $V_2[0, x_2] = 1/2$. Without loss of generality, assume $x_1 < x_2$. In this case, the proportional division of the cake is as follows:

Kid 1 receives the portion $[0, x_1]$.
Kid 2 receives the portion $[x_1, 1]$.

This division works because $V_1[0, x_1] = 1/2$ and $V_2[x_1, 1] \geqslant 1/2$.

For the other case, solution is similar.

Solution 2: We can solve the problem using 1 cut and 1 eval query as follows:

i. $x_1 = \texttt{cut}(1, 0, 1/2)$

ii. $v = \texttt{eval}(2, 0, x_1)$

Without loss of generality assume $v \leqslant 1/2$. In this case, the proportional division of the cake is as follows:

Kid 1 receives the portion $[0, x_1]$.
Kid 2 receives the portion $[x_1, 1]$.

In the other case, the division will be swapped between kids.

b) Yes, the algorithm is proportional. At each iteration $1, ..., n-1$ each agent receives a cake piece of value $\frac{1}{n}$. Now if we show that for iteration n, the last agent also receives a piece of value $\geqslant \frac{1}{n}$, then we are done. We will prove a stronger claim using the loop invariant.

Loop Invariant: After i iterations, the cake has value atleast $\frac{n-i}{n}$ for all the remaining $n - i$ kids.

- **Intialization:** Initially the cake value for all the kids is 1. Hence, the invariant is satisfied initially.
- **Iteration:** At iteration $i - 1$, the cake value for all the kids is at least $\frac{n-i+1}{n}$. Suppose the remaining cake interval is $[y, n]$ and kid j has the leftmost mark in the remaining cake in iteration i. So, for all the remaining kids except kid j, $V[y, x_j] \leqslant \frac{1}{n}$. This implies that for all the remaining kids except kid j, $V[x_j, n] \geqslant \frac{n-i+1}{n} - \frac{1}{n} = \frac{n-i}{n}$. Hence, the invariant is satisfied.
- **Termination:** After all the kids have been distributed cake pieces, the cake value is 0. Hence, the invariant is satisfied after termination.

Number of queries in worst case: In each round, every agent makes one cut query to determine the point where their value for the piece is $\frac{1}{n}$ of the total cake. After one agent receives their share, the algorithm repeats for the remaining $n - 1$ agents.

i. In the first round, there are n agents, so n cut queries are made.

ii. In the second rount, there are $n-1$ agents, so $n-1$ cut queries.

iii. This continues until only 1 agent is left.

Total Number of queries = $n + (n-1) + ... + 1 = \frac{n(n+1)}{2} = O(n^2)$

c) **Informal Idea:** We try to divide the range $[a, b]$ into two parts $[a, x]$ and $[x, b]$ such that for half of the kids the cake can be distributed fairly in the part $[a, x]$ and remaining in the part $[x, b]$. This can be done by querying point x_i for each kid i such that $V_i[a, x_i] = \frac{(b-a)}{2}$ and taking median of all x_i. Now we can divide $[a, b]$ into ranges $[a, x_k]$ and $[x_k, b]$ such that for all kids with $x_i \leqslant x_k$, $V_i[a, x_k] \geqslant \frac{(b-a)}{2}$ and for remaining kids $V_i[x_k, b] \geqslant \frac{(b-a)}{2}$. For base cases $n = 1$ return the whole range and $n = 2$ we can use solution to part (a).

Note: For simplicity, we assume that n is a power of 2.

ALGORITHM 1: Proportional Cake Division Algorithm

Input: Range $[a, b]$, number of kids n
Output: Proportional division of the cake among the kids

```
Function CakeDivision(a, b, n, kids):
    if n == 1 then
        Assign interval [a, b] to kids[1]
    else if n == 2 then
        x_1 ← CutQuery(kids[1], a, (b − a)/2)
        x_2 ← CutQuery(kids[2], a, (b − a)/2)
        if x_1 < x_2 then
            Assign interval [a, x_1] to kids[1]
            Assign interval [x_1, b] to kids[2]
        else
            Assign interval [a, x_2] to kids[2]
            Assign interval [x_2, b] to kids[1]
    else
        Let x[n] = [] // Array to store the cut points for
           each kid

        for each kid i in kids do
            x[i] ← CutQuery(kids[i], a, (b − a)/2)
        Let x_k be the median of array x // Since x has even
           length, there will be two medians.  We take
           the lower one
        Let left_kids = []
        Let right_kids = []
        for each kid i in kids do
            if x[i] ⩽ x_k then
                left_kids.append(i)
            else
                right_kids.append(i)
        CakeDivision(a, x_k, n/2, left_kids)
        CakeDivision(x_k, b, n/2, right_kids)
CakeDivision(0, 1, n, {1, 2, ..., n})
```

Proof of Correctness: We prove the above algorithm provides a proportional division of the cake using strong induction:

For $n = 1$ algorithm trivially gives the proportional answer.

Inductive Hypothesis $(P(n))$: *CakeDivision(a, b, n, kids)* returns a proportional division of the cake among the $n \geqslant 2$ kids such that each kid receives at least $\frac{c}{n}$ of the cake where $c = \min\{\texttt{eval(i,a,b)} \mid i \in \{1, 2, \ldots, n\}\}$.

Base Case: For $n = 2$, the algorithm queries both kids to determine their cuts x_1

and x_2, where they each believe the midpoint is. The algorithm then considers two cases:

i. If $x_1 < x_2$, the algorithm assigns the interval $[a, x_1]$ to kid 1 and $[x_1, b]$ to kid 2.

ii. If $x_2 < x_1$, the algorithm assigns the interval $[a, x_2]$ to kid 2 and $[x_2, b]$ to kid 1.

In either case, each kid receives a portion of the cake that they value as at least half of the total cake. Since $n = 2$, each kid values their piece as at least $\frac{c}{2}$, where $c = \min\{V_1[a, b], V_2[a, b]\}$. Hence, $P(2)$ holds.

Inductive Step: Assume that $P(k)$ holds for all $k < n$, i.e., for any number of kids $k < n$, the algorithm provides a proportional division where each kid receives at least $\frac{c}{k}$ of the cake, where $c = \min\{V_i[a, b] \mid i \in \{1, 2, \ldots, k\}\}$.

Now, consider the problem for n kids. The algorithm performs the following steps:

i. The algorithm queries each kid i to find the point x_i where they believe the interval $[a, x_i]$ is half the total value of the interval $[a, b]$.

ii. The algorithm selects the median value x_k from the set $\{x_1, x_2, \ldots, x_n\}$. This median split has the following properties:

   A. For $\frac{n}{2}$ kids, the left portion $[a, x_k]$ contains at least half of their valuation of the cake.

   B. For the remaining $\frac{n}{2}$ kids, the right portion $[x_k, b]$ contains at least half of their valuation of the cake.

iii. The algorithm then recurses on the left portion $[a, x_k]$ with the $\frac{n}{2}$ kids whose cuts $x_i \leqslant x_k$, and on the right portion $[x_k, b]$ with the $\frac{n}{2}$ kids whose cuts $x_i > x_k$.

Let $c = \min\{V_i[a, b] \mid i \in \{1, 2, \ldots, n\}\}$ be the minimum valuation of the interval $[a, b]$ among all kids. By the properties of the median cut, the subproblems have the following characteristics:

i. The left subproblem involves $\frac{n}{2}$ kids and in which each kid i believes that $V_i[a, x_k] \geqslant \frac{c}{2}$. By inductive hypothesis, each of these kids will receive at least $\frac{c/2}{n/2} = \frac{c}{n}$.

ii. The right subproblem involves $\frac{n}{2}$ kids and in which each kid i believes that $V_i[x_k, b] \geqslant \frac{c}{2}$. By inductive hypothesis, each of these kids will receive at least $\frac{c/2}{n/2} = \frac{c}{n}$.

Therefore, the algorithm ensures that each kid receives a portion of the cake of value at least $\frac{c}{n}$.

Since $c = 1$ for the whole cake, we get that the algorithm is proportional.

Number of queries: In each function call we are making $O(n)$ queries. Therefore the total number of queries will be:

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$$

$$T(n) = O(n \log n)$$

Note: To compute the median of an array of length n, one could sort the array in $\mathcal{O}(n \log n)$ time and select the middle value. There is a more sophisticated $\mathcal{O}(n)$ time algorithm known for this problem. However, since we are concerned with the *number* of queries and not the *time* taken by the algorithm, the time taken to find a median is not relevant to our analysis.

d) The solution for part (c) does not use *eval* queries.

2. Consider the "Twenty Questions" game. In this game, the first player thinks of a number in the range 1 to n. The second player has to figure out this number by asking the fewest number of true/false questions. Assume that all responses are honest.

   a) What is an optimal strategy if n is known?

   b) What is a good strategy if n is not known?

An optimal strategy to identify the number in the range 1 to n is to use the **binary search algorithm**. The steps of the binary search algorithm are as follows:

a) Let the range of possible numbers be from 1 to n.

b) While the range contains more than one number:

- Compute the middle point m of the current range: $m = \left\lfloor \frac{\text{low}+\text{high}}{2} \right\rfloor$.
- Ask if the number is greater than m.
  - If the answer is **yes**, then set the new range to $m + 1$ to high.
  - If the answer is **no**, then set the new range to low to m.

c) Once the range is reduced to a single number, that number is the answer.

The binary search algorithm requires $O(\log n)$ queries to determine the correct number.

Proof of Optimality:
To prove the optimality of the binary search strategy, consider the following argument from information theory:

- To uniquely identify a number in the range 1 to n, we need to distinguish between n possible outcomes.

- The binary representation of the number requires $\lceil \log_2 n \rceil$ bits since each bit can be thought of as a true/false answer to a binary question.
- Each true/false question in the game corresponds to revealing one bit of the binary representation of the number.
- Therefore, in the worst case, we need at least $\log_2 n$ bits (or true/false questions) to uniquely identify the number.
- If we could find the number in fewer than $\log_2 n$ questions, this would imply that fewer than $\log_2 n$ bits could represent all n possible numbers, which is impossible as this would violate the basic principles of information theory.
- Hence, $O(\log n)$ queries are necessary and sufficient to identify the number.

(b) Strategy if n is Not Known

If n is not known, we can adopt an iterative approach:

a) Start by asking if the number is greater than $2^0 = 1$.

b) Continue asking if the number is greater than 2^k for increasing values of k (i.e., $1, 2, 4, 8, \ldots$) until the answer is **no**.

c) Once you find 2^k such that the number is not greater than 2^k, you know the number lies between $2^{k-1} + 1$ and 2^k.

d) Apply the binary search algorithm on the range $2^{k-1} + 1$ to 2^k to determine the exact number.

Time Complexity
This strategy requires $O(\log m)$ queries, where m is the actual number, which is unknown at the start.
There are two parts in the algorithm. First part is finding k such that $2^{k-1} \leqslant m < 2^k$ and then second part is finding the actual m doing a binary search in the range $[2^{k-1}, 2^k]$. We know that $2^{k-1} \leqslant m < 2^k$ and hence $k = \lfloor \log_2 m \rfloor + 1$. It would take $O(k)$ queries to find k. In the second part, we do a binary search in the range $[2^{k-1}, 2^k]$ and hence the number of queries required for the same is $O(\log_2(2^{k-1})) = O(k)$. Total number of queries are $O(k) = O(\lfloor \log_2 m \rfloor)$.

a)
- Written "I do not know how to approach this problem" - 0.3 points
- – High level idea of Binary search - 0.5 points
- – High level idea of $\Omega(\log n)$ lower bound - 1 point

b)
- Written "I do not know how to approach this problem" - 0.3 points

- – Correct idea - 1 point
  – Showing it requires $O(\log n)$ queries - 0.5 point

3. In class, we saw an $\mathcal{O}(n \log n)$ time algorithm to count the number of inversions in an array $A[1:n]$. Recall, an inversion is a pair (i, j) with $1 \leqslant i < j \leqslant n$ and $A[i] > A[j]$. Let us call an inversion (i, j) *bounded* if $A[j] < A[i] \leqslant A[j] + 10$. That is, $A[i]$ is bigger than $A[j]$ but not by much. Design an $\mathcal{O}(n \log n)$ time algorithm to count the number of bounded inversions. You may assume that the numbers in the array A are distinct.

ALGORITHM 2: `CountBoundedInversions(`A`, `$left$`, `$right$`)`

Input: Array $A[1:n]$ and indices $left, right$.
Output: Number of bounded inversions in $A[left \ldots right]$.

```
1  if left ⩾ right then
2      return 0
3  mid ← (left + right)/2
4  count ← 0
5  count ← CountBoundedInversions(A, left, mid)
6  count ← count + CountBoundedInversions(A, mid + 1, right)
7  i ← left
8  j ← mid + 1
9  Create empty array temp
10 while i ⩽ mid and j ⩽ right do
11     if A[i] > A[j] then
12         count_all ←
             count of elements in left subarray greater than A[j]
13         count_shifted ←
             count of elements in left subarray greater than A[j] + 10
14         count ← count + (count_all − count_shifted)
15         Add A[j] to temp array and increment j
16     else
17         Add A[i] to temp array and increment i
18 while i ⩽ mid do
19     Add remaining elements from left subarray to temp
20     i ← i + 1
21 while j ⩽ right do
22     Add remaining elements from right subarray to temp
23     j ← j + 1
24 Copy temp array back to A[left ... right]
25 return count
```

Proof of Correctness:
The algorithm uses a divide-and-conquer approach similar to the classic inversion counting algorithm. The main difference lies in how the inversions are counted.

- **Base Case:** - When the subarray has one or no elements, there are no inversions, so the algorithm correctly returns 0.
- **Recursive Case:** - The array is split into two halves, and the function recursively counts bounded inversions in both halves. Since these recursive calls are correct by the inductive hypothesis, they return the correct number of bounded inversions within each half.
- **Counting Cross-Inversions:** - After counting the inversions within each half, the algorithm counts the bounded inversions that involve one element from the left half and one from the right half (i.e., cross-inversions). - For each element $A[j]$ in the right subarray, the algorithm counts how many elements $A[i]$ in the left subarray satisfy $A[j] < A[i]$ (i.e., count_all) and how many elements 'A[i]' in the left subarray satisfy $A[j] + 10 < A[i]$ (i.e., count_shifted). - The difference between 'count_all' and 'count_shifted' gives the number of bounded inversions for that specific pair (i, j).
- **Merge Step:** - The merge step involves combining the two sorted halves while maintaining the order, similar to the merge step in merge sort. This ensures that the algorithm runs in 'O(n log n)'*time*.

- **Correctness of Bounded Inversion Counting:** - The algorithm correctly counts the bounded inversions by carefully tracking only those pairs that satisfy the condition $A[j] < A[i] \leqslant A[j] + 10$. - This approach guarantees that all bounded inversions are counted, as each potential inversion is considered during the merging of the left and right subarrays.

  By using this divide-and-conquer strategy and handling bounded inversions during the merge step, the algorithm ensures that it correctly counts all bounded inversions while maintaining the overall $O(n \log n)$ time complexity.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct algorithm idea - 1.5 points
  - Correct proof idea - 1 point
  - Proving $O(n \log n)$ time complexity - 0.5 points

4. We are given n wooden sticks, each of integer length, where the i^{th} piece has length $L[i]$. We seek to cut them so that we end up with k pieces of exactly the same length, in addition to other fragments. Furthermore, we want these pieces to be as large as possible.

   a) Given four wooden sticks of lengths $L = [10, 6, 5, 3]$, what are the largest sized pieces you can get for $k = 4$? (Hint: The answer is not 3.)

b) Design a polynomial-time algorithm that, for a given L and k, returns the maximum possible length of the k equal pieces cut from the original n sticks.

a) The largest size of the pieces that can be obtained is 5. Specifically, you can cut the sticks into four pieces of length 5 (cutting the stick of length 10 into two pieces of length 5 each).

b) To solve this problem, we can use binary search on the answer. The idea is as follows:

- **Step 1:** Define the search interval for the possible lengths of the pieces. The smallest possible length of each piece is $\frac{1}{k}$ (since minimum length of stick is 1, which may need to be divided into k pieces), and the largest possible length is given by $\frac{\max(L)}{k}$. Note that any consecutive points in the binary search space will be $\frac{1}{k}$ distance apart. So, the total number of points in our search space will be $O(k * \frac{\max(L)}{k}) = O(\max(L))$.
- **Step 2:** Perform a binary search within this interval. Let the $\lfloor x \rfloor_{\frac{1}{k}}$ function represent the largest multiple of $\frac{1}{k}$ less than or equal to x. Then, for each mid-point length $\ell = \lfloor \frac{lower+upper}{2} \rfloor_{\frac{1}{k}}$ in the binary search:
  - Check if it's possible to obtain k pieces of length ℓ by summing the number of pieces we can obtain from each stick i as $\lfloor L[i]/\ell \rfloor$, where $\lfloor x \rfloor$ is the greatest integer function.
  - If the total number of pieces is greater than or equal to k, then ℓ is a candidate for the maximum length. Move the lower bound of the binary search to $\ell + \frac{1}{k}$. [$lower = \ell + \frac{1}{k}$]
  - Otherwise, move the upper bound of the binary search to $\ell - \frac{1}{k}$. [$upper = \ell - \frac{1}{k}$]
- **Step 3:** Continue this process until $lower \geqslant upper$. The result is the maximum possible length of the pieces.

Time Complexity: Since there are $O(\max(L))$ points in the search space, the binary search can take up to $O(\log(\max(L)))$ iterations. It can be seen that each iteration takes $O(n)$ time (assuming division is $O(1)$) and hence, the total time taken by the above algorithm is $O(n \log(\max(L)))$.

a)
- Written "I do not know how to approach this problem" - 0.2 points
- Correct Answer (5) - 1 point

b)
- Written "I do not know how to approach this problem" - 0.4 points
- **Note:** Although the final answer may not be integral, for the purpose of grading, no marks will be deducted for assuming that the maximum

possible length should be integral.

- Correct binary search idea - 1.5 points
- Showing $O(n \log(\max(L)))$ time complexity - 0.5 points

5. Suppose an undirected graph $G = (V, E)$ is represented both as an adjacency matrix and an adjacency list. What is the time complexity of the following operations under each of these representations?

   a) Checking whether two vertices u and v are adjacent in G.

   b) Computing the total number of vertices adjacent to the vertex u.

   c) Computing the number of common neighbors of two vertices u and v.

   d) Adding or removing a vertex or an edge.

   Which of your answers would change, and how, if you additionally knew whether the graph was sparse or dense?

a)
- **Adjacency Matrix:** $O(1)$. We can directly check the entry at the matrix position $[u][v]$.
- **Adjacency List:** $O(\deg(u))$. We need to traverse the list of neighbors of u to see if v is a neighbor.

b)
- **Adjacency Matrix:** $O(|V|)$. We need to scan the entire row corresponding to u in the matrix.
- **Adjacency List:** $O(\deg(u))$, i.e., the number of adjacent vertices is the size of the list of neighbors of u. This is because we need iterate over the list of neighbours of u.

c)
- **Adjacency Matrix:** $O(|V|)$. We need to scan the rows for both u and v and count the number of columns where both rows contain 1.

  Note: There is a nice trick to compute the number of common neighbors between *every* pair of vertices in a single stroke: Just multiply the adjacency matrix with itself! The $(i, j)^{\text{th}}$ entry of the matrix product gives the number of common neighbors between the vertices v_i and v_j. Can you see why?
- **Adjacency List:**

  The time taken would be $O(\deg(u) \times \deg(v))$ as then we need to check for each node pair in the two lists if they are equal.

  Finding intersection in $O(\deg(u) + \deg(v))$: If the nodes are maintained in a sorted order (which is usually not the case) in the adjacency list, then we can do the following:

Initiation: Keep two pointers, one for each list, initiated at the start of the list.
Step 1: At any point, check the nodes at both pointers. If they are the same, then increment the answer by 1 as we found a common neighbor. Also, increment both pointers.
Step 2: If the two pointers do not have the same node, then we increment the pointer that has a smaller node (smaller with respect to the ordering that has been used to sort the nodes in a list) and repeat until any of the list ends.
Time: We iterated each element in two lists at most once. And hence the time taken is $O(\deg(u) + \deg(v))$.

d)
- **Adjacency Matrix:**
  - **Add/Remove Vertex:** $O(|V|)$. This is because adding a vertex amounts to adding an extra row and an extra column to the existing matrix.
  - **Add/Remove Edge:** $O(1)$. Simply update the matrix entry.
- **Adjacency List:**
  - **Add/Remove Vertex:** $O(1)$ for adding a vertex, but $O(|V| + |E|)$ if we need to remove a vertex as it requires updating all lists.
  - **Add/Remove Edge:** $O(1)$ for adding an edge; $O(\deg(u) + \deg(v))$ for removing an edge (u, v) if implemented as a simple list.

Sparse vs. Dense Graph:

- For sparse graphs (where $|E|$ is much less than $|V|^2$), the adjacency list is generally more space and time-efficient, especially for operations like checking adjacency or listing neighbors.
- For dense graphs (where $|E|$ is close to $|V|^2$), the adjacency matrix becomes more efficient, particularly for checking adjacency and finding common neighbors since the operations are $O(1)$ and $O(|V|)$ respectively, which are better than the corresponding operations in the adjacency list.

6. A *bipartite* graph is a graph $G = (V, E)$ whose vertices can be partitioned into two sets (i.e., $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set. Give a linear-time algorithm to determine whether a given graph is bipartite.

Informal Idea: A graph is bipartite if and only if it does not contain any odd-length cycles. We can verify this property using a BFS traversal. The idea is to determine the depth parity (even or odd) of each vertex during the traversal. We start by assigning the root vertex a depth parity of 0. As we explore the graph using BFS, each neighbor

is assigned a depth parity opposite to that of the current vertex. If we encounter a neighbor that has already been assigned a parity and it matches the parity of the current vertex, this indicates the presence of an odd-length cycle, meaning the graph is not bipartite.

ALGORITHM 3: Check if Graph is Bipartite (BFS Approach)

Input: Graph $G(V, E)$
Output: True if the graph is bipartite, False otherwise

```
1  Function BipartiteCheck(G):
2      Initialize a depth_parity array depth_parity[V] with −1 (unvisited) for all vertices
3      foreach vertex v ∈ V do
4          if depth_parity[v] = −1 then
5              if NOT BFS(v, G, depth_parity) then
6                  return False
7      return True
8  Function BFS(v, G, depth_parity):
9      Initialize a queue Q
10     depth_parity[v] ← 0
11     Add vertex v to Q
12     while Q is not empty do
13         Remove a vertex u from front of the Q
14         foreach neighbor n of u in G do
15             if depth_parity[n] = −1 then
16                 depth_parity[n] ← 1 − depth_parity[u]
17                 Add vertex n to the Q
18             else if depth_parity[n] = depth_parity[u] then
19                 return False
20     return True
```

Lemma: If there is an odd closed walk in a graph, then there is an odd length cycle.

Proof: We will prove by strong induction on the number of edges k in the closed walk.
Base Case: The base case $k = 1$, when the closed walk is a loop, holds trivially.
Inductive Step: Suppose induction hypothesis is true for $k \leqslant 2r - 1$ where r>1. Let $W = (w_1, w_2, ..., w_{2r+1}, w_1)$ be a closed walk of $2r + 1$ edges. If all vertices in the walk are distinct, then we have a cycle of length $2r + 1$. Otherwise there exists (i, j), $1 \leqslant i \leqslant j \leqslant 2r + 1$ such that $w_i = w_j$. Then W can be written as $(w_1, ..., w_i, ..., w_j, ..., w_1)$ and therefore $W' = (w_1, ..., w_i, w_{j+1}, .., w_{2r+1}, w_1)$ is an odd closed walk of length $\leqslant 2r - 1$. By strong induction on W' there exists a odd cycle.

Claim: A graph $G = (V, E)$ is bipartite if and only if there are no odd-length cycles present in the graph G.

Proof: ($\Rightarrow$) If $G = (V, E)$ is bipartite with vertex set V_1 and V_2, every step along a walk

takes a step either from V_1 to V_2 or from V_2 to V_1. So every closed walk from a vertex $v \in V$ will be of even length. Therefore every cycle will be of even length.
($\Leftarrow$) Let $v_0 \in V$ be an arbitrary vertex in $G = (V, E)$. Then consider the following two vertex sets:

$$V_1 = \{v \in V \mid d(v, v_0) \text{ is even}\}$$

$$V_2 = \{v \in V \mid d(v, v_0) \text{ is odd}\}$$

$$d(u, v) \text{ is shortest distance between vertices } u \text{ and } v$$

Now we will show that the above sets satisfy bipartite conditions. $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$ are trivial.
Suppose that there is an edge between x and y where $x, y \in V_1$ or $x, y \in V_2$. Let the shortest path between v_0 and x be $(v_0, ..., x)$ and v_0 and y be $(v_0, ..., y)$. Then there is closed walk of odd length $(v_0, ..., x, y, ..., v_0)$. By lemma above G contains an odd cycle, which is contradiction. Hence graph G is a bipartite graph.

Time Complexity: $O(|E| + |V|)$.

7. ($\star$) Let X and Y be two independent random variables whose domain is $\{0, 1, 2, ..., n\}$. You are given their explicit distributions $(p_0, p_1, ..., p_n)$ and $(q_0, q_1, ..., q_n)$ where $p_i = \Pr[X = i]$ and $q_j = \Pr[Y = j]$. Let $Z = X + Y$ be the sum of these two random variables. Design an algorithm that outputs the distribution of Z; that is, for all k you should be able to read out $\Pr[Z = k]$. The running time of your algorithm should be much faster than $\Theta(n^2)$.

Informal Idea: We will use the Karatsuba algorithm to solve this problem. Consider the following polynomials:

$$\begin{aligned} P &= p_n x^n + p_{n-1} x^{n-1} + \cdots + p_0, \\ Q &= q_n x^n + q_{n-1} x^{n-1} + \cdots + q_0, \\ R &= P \times Q = r_{2n} x^{2n} + r_{2n-1} x^{2n-1} + \cdots + r_0. \end{aligned}$$

We can observe that $\Pr[Z = k] = r_k$.
We can express P and Q as:

$$\begin{aligned} P &= x^{\lceil \frac{n}{2} \rceil} P_1 + P_2, \\ P_1 &= p_n x^{\lfloor \frac{n}{2} \rfloor} + p_{n-1} x^{\lfloor \frac{n}{2} \rfloor - 1} + \cdots + p_{\lfloor \frac{n}{2} \rfloor + 1}, \\ P_2 &= p_{\lfloor \frac{n}{2} \rfloor} x^{\lfloor \frac{n}{2} \rfloor} + p_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1} + \cdots + p_0. \end{aligned}$$

Similarly,

$$
\begin{aligned}
Q &= x^{\lceil \frac{n}{2} \rceil} Q_1 + Q_2, \\
Q_1 &= q_n x^{\lfloor \frac{n}{2} \rfloor} + q_{n-1} x^{\lfloor \frac{n}{2} \rfloor - 1} + \cdots + q_{\lfloor \frac{n}{2} \rfloor + 1}, \\
Q_2 &= q_{\lfloor \frac{n}{2} \rfloor} x^{\lfloor \frac{n}{2} \rfloor} + q_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1} + \cdots + q_0.
\end{aligned}
$$

Now, the product R is given by:

$$
\begin{aligned}
R &= (x^{\lceil \frac{n}{2} \rceil} P_1 + P_2) \times (x^{\lceil \frac{n}{2} \rceil} Q_1 + Q_2) \\
&= x^{2\lceil \frac{n}{2} \rceil} P_1 Q_1 + x^{\lceil \frac{n}{2} \rceil} (P_1 Q_2 + P_2 Q_1) + P_2 Q_2 \\
&= x^{2\lceil \frac{n}{2} \rceil} R_1 + x^{\lceil \frac{n}{2} \rceil} R_2 + R_3,
\end{aligned}
$$

where

$$
\begin{aligned}
R_1 &= P_1 Q_1, \\
R_3 &= P_2 Q_2, \\
R_2 &= P_1 Q_2 + P_2 Q_1 = (P_1 + P_2)(Q_1 + Q_2) - R_1 - R_3.
\end{aligned}
$$

Thus, we can find $P_1 Q_1$, $P_2 Q_2$, and $(P_1 + P_2)(Q_1 + Q_2)$ independently and finally combine their results to obtain R, as discussed in class (Karatsuba's Algorithm).

Time Complexity:

$$T(n) = 3 \cdot T(\frac{n}{2}) + O(n)$$

By Master theorem, $T(n) = O(n^{\log_2 3})$