National University of Computer and Emerging Sciences

Karachi Campus

Design and Analysis of Algorithms (CS2009)

Sessional-II Exam

Total Time (Hrs): 1.5

Total Marks: 17.5

Total Questions: 5

Date: Nov,5th 2024

Course Instructor(s)

Dr.Muhammad Atif Tahir, Dr.Nasiruddin, Dr.Kamran, Dr. Fahad Sherwani, Dr. Farrukh Salim, Ms. Anaum Hamid,Mr.Syed Faisal Ali,Mr.Sandesh,Mr.Minhal Raza ,Mr. Abu Zohran

Roll No _____ Section _____ Student Signature _____

Do not write below this line

Attempt all the questions.

CLO #1: To apply acquired knowledge to solve computing problems complexities and proofs.

Q1 [3 Points]: Given the text T="abcabcababcabcabc" and the pattern P="abcabcab", use the **Knuth-Morris-Pratt (KMP) algorithm modified version given below** to determine all occurrences of P in T. Show the working of KMP including the complete failure (prefix) table and perform a step-by-step dry run of the matching process. Document each pointer movement and skip, particularly when using the failure table after mismatches

```
Algorithm KMPMatchAll(T, P)
    F ← failureFunction(P) // Preprocess pattern P to create the failure function F (Refer to Appendix)
    i ← 0 // Index for text T
    j ← 0 // Index for pattern P
    n ← length(T) // Length of the text T
    m ← length(P) // Length of the pattern P
    matches ← [ ] // List to store starting indices of matches
    while i < n
        if T[i] = P[j] then
            if j = m - 1 then
                matches.append(i - j) // Store the starting index of the match
                j ← F[j] // Update j using the failure function for next potential match
            else
                i ← i + 1 // Move to the next character in the text
                j ← j + 1 // Move to the next character in the pattern
        else
            if j > 0 then
            j ← F[j - 1] // Use the failure function to find the next position in the pattern
                else
                    i ← i + 1 // Move to the next character in the text
    return matches // Return the list of starting indices of all matches
```

```
Algorithm failureFunction(P) // for Question-1
        F[0] = 0
        i = 1
        j = 0
while i < m
        if P[i] = P[j]
                {we have matched j + 1 chars}
                F[i] = j + 1
                i = i + 1
                j = j + 1
        else if j > 0 then
                {use failure function to shift P}
                j = F[j - 1]
        else
                F[i] = 0 { no match }
                i = i + 1
```

Solution :

Step 1: Construct the KMP Failure Table

For P="abcabcab":

1. Initialize the failure array F as [0,0,0,0,0,0,0,0].

2. Set j=0 to track the longest prefix-suffix match.

i = 1: P[1]='b' does not match 'P[0]='a', so F[1]=0.
i = 2: P[2]='c' does not match P[0]='a', so F[2]=0.
i = 3: P[3]='a' matches P[0]='a', increment j=1, F[3]=1.
i = 4: P[4]='b' matches P[1]='b', increment j=2, F[4]=2.
i = 5: P[5]='c' matches P[2]='c', increment j=3, F[5]=3.
i = 6: P[6]='a' matches P[3]='a', increment j=4, F[6]=4.
i = 7: P[7]='b' matches P[4]='b', increment j=5, F[7]=5.

The failure table F is:
F=[0,0,0,1,2,3,4,5]

Step 2: Perform KMP Matching

1. Start with i=0, j=0, and use the failure table to jump j back only when a mismatch occurs.

2. Matching Process:

i=0 to i=7: Characters match at each position, so j advances to 8. Pattern P is found at i−j=0. Reset j=F[7]=5.

i=8, j=5: Continue matching from position j=5 in P, matches until i=15, giving a match at i−j=8.

CLO #4 : To construct and analyze real world problems solutions using different algorithms design techniques i.e. Brute Force, Divide and Conquer, Dynamic Programming, Greedy Algorithms.

Q2 [3 Points]: Design an algorithm that calculates the n-th Fibonacci number in linear time using dynamic programming. Clearly describe the main steps of Dynamic Programming for the above mentioned of nth Fibonacci series. You may design the algorithm using the main principles of Dynamic Programming including (i) the structure of an optimal solution (ii) Principle of Optimality (iii) Bottom Up Computation (iv) (Optional) Construction of any optimal solution. Remember, The Fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers. For example, 0, 1, 1, 2, 3, 5, 8 and so on

Solution:

To design an algorithm that calculates the n-th Fibonacci number in linear time using dynamic programming, we'll break down the process according to the principles of dynamic programming.

Problem Definition

The Fibonacci sequence $F(n)$ is defined as follows:

$$F(0) = 0, \quad F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n \geq 2$$

Our goal is to compute $F(n)$ in linear time, making use of dynamic programming principles.

Dynamic Programming Principles Applied

1. **Structure of an Optimal Solution** The n-th Fibonacci number can be computed from the $(n-1)$-th and $(n-2)$-th Fibonacci numbers. This relationship provides the structure of our optimal solution:

   $$F(n) = F(n-1) + F(n-2)$$

   This shows that if we know the two preceding Fibonacci numbers, we can easily compute the next one. The solution thus exhibits an overlapping subproblem structure.

2. **Principle of Optimality** To compute $F(n)$, we only need the values of $F(n-1)$ and $F(n-2)$. This means that the solution to $F(n)$ can be constructed optimally by solving the smaller subproblems $F(n-1)$ and $F(n-2)$ first. Since each Fibonacci number only depends on the previous two numbers, we don't need to recompute any values, and each subproblem can be solved once and reused.

3. **Bottom-Up Computation** Instead of starting from $F(n)$ and recursively working down, we start from the smallest subproblems $F(0)$ and $F(1)$, and iteratively build up to $F(n)$. This avoids the exponential time complexity of recursive solutions and ensures linear time complexity $O(n)$.

4. **(Optional) Construction of an Optimal Solution** Although the Fibonacci sequence itself doesn't require reconstructing a solution path, we'll use an iterative approach to compute $F(n)$ efficiently.

Algorithm Design

We will use an iterative, bottom-up approach and store only the last two computed Fibonacci numbers to save space.

Algorithm Steps:

1. **Initialize Base Cases:** Start by initializing the values for $F(0)$ and $F(1)$.
2. **Iterate Up to n:** For each i from 2 to n, compute $F(i)$ using $F(i) = F(i-1) + F(i-2)$.
3. **Store Only Last Two Values:** Instead of storing all computed Fibonacci numbers, keep track of only the last two numbers, updating them as we progress through the sequence.
4. **Return Result:** Once $i = n$, return the computed $F(n)$.

Pseudocode

Here's the pseudocode for the algorithm:

```
python


  function Fibonacci(n):
      if n == 0:
          return 0
      elif n == 1:
          return 1

      # Initialize base cases
      prev2 = 0   # F(0)
      prev1 = 1   # F(1)

      # Bottom-up calculation
      for i = 2 to n:
          current = prev1 + prev2
          prev2 = prev1
          prev1 = current

      # F(n) will be in prev1
      return prev1
```

Explanation of the Algorithm:

- **Initialization:** If $n = 0$ or $n = 1$, return the respective Fibonacci number directly (base cases).
- **Iteration:** Starting from $i = 2$, compute each Fibonacci number up to $F(n)$ by adding the previous two numbers. After each computation, update the stored values to move one step forward in the sequence.
- **Result:** Once the loop finishes, the variable `prev1` will contain $F(n)$.

Complexity Analysis

- **Time Complexity:** $O(n)$ because we compute each Fibonacci number once up to n.
- **Space Complexity:** $O(1)$ since we only store the last two Fibonacci numbers at any given time.

Example Walkthrough

For $n = 5$:

Initialize $\text{prev2} = 0, \text{prev1} = 1$.
$i = 2$: current = 1 (0 + 1), update $\text{prev2} = 1, \text{prev1} = 1$.
$i = 3$: current = 2 (1 + 1), update $\text{prev2} = 1, \text{prev1} = 2$.
$i = 4$: current = 3 (1 + 2), update $\text{prev2} = 2, \text{prev1} = 3$.
$i = 5$: current = 5 (2 + 3), update $\text{prev2} = 3, \text{prev1} = 5$.

Finally, the result $F(5) = 5$ is returned.

CLO #3 : To evaluate generic algorithmic solutions such as sorting, searching and graphs applied to real-world problems

Q3 [3 Points]: Create a list of tuples or pairs, where each tuple contains a player ID and their corresponding score. For example, player_scores = [(ID_1, $score_1$), (ID_2, $score_2$), ..., (ID_n, $score_n$)]. This is a multiplayer online game, where each player is assigned a unique score based on their performance. The game developers need to display the top 100 players' scores quickly and in linear time. The scores are represented as 32-bit integers ranging from 0 to 1,000,000. Design an algorithm (pseudocode) to sort and display the top 100 scores from a list of player scores. You are also required to design algorithm that use memory as efficiently as possible and consider implementing an additional step to keep track of the player IDs associated with each score. Explain your approach in pseudocode and the potential benefits of your modifications.

Solution :

Step-by-Step Algorithm

Input Data:
1. Create a list of tuples or pairs, where each tuple contains a player ID and their corresponding score. 2. Example: player_scores = [(ID1, score1), (ID2, score2), ..., (IDn, score_n)]

Identify the Maximum Score:

a) Traverse the list of scores to find the maximum score to determine the number of digits (k) needed for sorting.
b) max_score = max(score for ID, score in player_scores)

Implement the Counting Sort:

Create a function counting_sort(arr, exp) that sorts the array based on the digit represented by exp (1s, 10s, 100s, etc.):

1. Initialize an output list output of the same size as arr to hold sorted values.
2. Create a count array of size 10 (for digits 0-9) and initialize all values to 0.

3. For each player score in arr, update the count array based on the current digit value.
4. Modify the count array to reflect actual positions in the output array.
5. Build the output array by placing elements in their correct positions based on the count array, ensuring to maintain the

Order of player IDs. Radix Sort Function:

i. Create the radix_sort(player_scores) function:

ii.Iterate over each digit (from least significant to most significant):Call counting_sort(player_scores, exp) for each digit.This ensures that the list is sorted based on all digits, resulting in a fully sorted list of playerscores.

Get Top 100 Scores:

After sorting, slice the sorted list to retrieve the top 100 player scores and their corresponding IDs: top_100 = sorted_player_scores[:100]

Display Results:

Loop through the top_100 list and print each player ID along with their score.

```
#include <stdio.h>

// Define a structure to hold player ID and score
typedef struct {
    int ID;
    int score;
} Player;

// Function to find the maximum score in player_scores
int find_max(Player player_scores[], int n) {
    int max = player_scores[0].score;
    for (int i = 1; i < n; i++) {
        if (player_scores[i].score > max) {
            max = player_scores[i].score;
        }
    }
    return max;
}

// Counting sort based on the digit represented by exp
void counting_sort(Player arr[], int n, int exp) {
    Player output[n]; // Output array to store sorted values
    int count[10] = {0}; // Count array to store occurrences of digits

    // Count occurrences of digits in the scores
    for (int i = 0; i < n; i++) {
        int index = (arr[i].score / exp) % 10;
        count[index]++;
    }

    // Accumulate counts
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Build output array based on current digit
    for (int i = n - 1; i >= 0; i--) {
        int index = (arr[i].score / exp) % 10;
        output[count[index] - 1] = arr[i];
        count[index]--;
    }

    // Copy sorted values back to the original
```

```
array
  for (int i = 0; i < n; i++) {
    arr[i] = output[i];
  }
}

// Radix sort on the player_scores array
void radix_sort(Player player_scores[], int n) {
  int max_score = find_max(player_scores, n);

  // Apply counting sort for each digit place (exp = 1, 10, 100, ...)
  for (int exp = 1; max_score / exp > 0; exp *= 10) {
    counting_sort(player_scores, n, exp);
  }
}

// Main function
int main() {
  // Example array of player scores (ID, score)
  Player player_scores[] = {
    {1, 502345}, {2, 100}, {3, 423432}, {4, 754321}, {5, 999999},
    {6, 235432}, {7, 543211}, {8, 345673}, {9, 12345}, {10, 876543}
  };
  int n = sizeof(player_scores) / sizeof(player_scores[0]);

  // Apply radix sort
  radix_sort(player_scores, n);

  // Extract and display the top 100 scores
  // Note: Here we print the top `n` scores since this example array is small
  printf("Top Scores:\n");
  for (int i = n - 1; i >= (n > 100 ? n - 10
```

CLO #4 : To construct and analyze real world problems solutions using different algorithms design techniques i.e. Brute Force, Divide and Conquer, Dynamic Programming, Greedy Algorithms.

Q4 [3.5 Points]: Imagine you're packing for a trip and need to fit everything you want to bring into a suitcase. There's only so much room in your suitcase, so you can't pack items that would collectively exceed the available space. Airlines have weight restrictions on luggage, so the combined weight of your suitcase and its contents must not go over a specific limit.

With these limitations, you'll need to prioritize. Some items, like a laptop or passport, are essential and they must be part of the suitcase while others are non-essential items, like multiple pairs of shoes, extra clothes, Camera, Shoes or Book, might be nice to have but less important. After fitting Laptop and Passport, you can only fit 8 kg of items in suitcase. The goal is to find non-essential items that provide the most "value" or importance to you for your trip, without exceeding your remaining luggage's weight limits of 8 kg per suitcase. For the above problem provide a solution based on dynamic programming bottom up approach to find out which pairs of non-essential items you are selected for packing the suitcase that gives you most importance value.

Item (i)	Weight (i)	Importance (i)
Book	**1**	**1**
Shoes	**4**	**5**
Camera	**3**	**4**
Clothes	**5**	**7**
Toilette's	**2**	**3**
Laptop	**6**	**9**
Passport	**7**	**11**

Solution :

This problem is an example of the Knapsack Problem. Here, we need to fit non-essential items into the suitcase in a way that maximizes their "importance" while staying within the weight limit of 8 kg. We'll solve it using a bottom-up dynamic programming (DP) approach to optimize the total importance.

Since the Laptop and Passport are essential, they must be packed, which leaves us with only 8 kg of weight allowance for additional non-essential items.

Define the Problem: We need to maximize the total importance of non-essential items without exceeding a weight limit of 8 kg.

Dynamic Programming Table

Let:

- $W = 8$: Remaining weight capacity after packing the essential items.
- **Items**: Only non-essential items are considered.

Item	Weight	Importance
Book	1	1
Shoes	4	5
Camera	3	4
Clothes	5	7
Toiletries	2	3

Step 1: DP Table Initialization

We create a DP table where the number of rows equals the number of items (5 items), and the number of columns equals the maximum weight (8 kg). The dimensions of the table are `(6 x 9)` because we also account for the base case (0 items and weight 0).

Step 2: DP Recurrence

The recurrence relation for the knapsack problem is:

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if item } i \text{ is not included} \\ \max(dp[i-1][w], dp[i-1][w - weight[i]] + importance[i]) & \text{if item } i \text{ is included} \end{cases}$$

Item (i)	Weight (i)	Value (i)	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1	1
2	4	5	0	1	1	1	5	6	6	6	6
3	3	4	0	1	1	4	5	6	6	9	10
4	5	7	0	1	1	4	5	7	8	9	**11**
5	2	3	0	1	3	4	5	7	8	10	**11**

Final Answer:

Maximum Importance: 11

Total Weight: 8 kg

Selected Items: Camera and Clothes

Conclusion:

Maximum Importance is 11, and the selected items fit within the 8 kg weight limit.

The total weight of selected items is 5+3 = 8 kg, which is within the allowed limit.

On selecting Item 4 which is Clothes we have max importance at this instance in table is =11.

Clothes own importance is 7 So , 11-7 =4 is our now Max Importance value

More over Max Allowed Weight is = 8 Kg per bag , So Upon Selecting Clothes which has weight of 5 kg Our Remaining Weight of Suitcase IS = 8 Kg - 5 Kg = 3 Kg

Now We have Remaining Max Importance is =4 and Remaining Max Weight is = 3kg

Finding 4 which our Max Remaining Importance in our table we observe that 4 is present against item 3 which is camera

So camera own importance is 4 So 4-4 =0

Max Importance value is become zero

Similarly Max Remaining Weight is = 3 Kg , So Upon Selecting Camera which has weight of 3 kg Our Remaining Weight of Suitcase IS = 3 Kg - 3 Kg = 0 Kg

Both the Max Importance and Max Weight is =0

So We Conclude That

Maximum Importance: 11

Total Weight: 8 kg

Selected Items: Camera and Clothes

Camera (3 kg, 4 importance) and Clothes **(5 kg, 7 importance)** add up to a total weight of 5 + 3 = 8 kg, and a total importance of 7+4=11.

CLO #1 : To apply acquired knowledge to solve computing problems complexities and proofs.

Q5 [5 Points]: You are an engineer tasked with optimizing network coverage across several cell towers in a large region. The towers are represented by their coordinates on a 2D grid as follows:
1. Tower P: (7, 1)
2. Tower Q: (1, 7)
3. Tower R: (3, 4)
4. Tower S: (4, 3)
5. Tower T: (2, 5)
6. Tower U: (5, 2)

Complete the following
(a) [1 Point] Sort all towers co-ordinates according to x-axis using Counting Sort? Show all steps clearly[1 Point]

Step 1: Identify the Range of the x-coordinates

The x-coordinates of the points are:

- $x_1 = 7$ (from $P_1(7, 1)$)
- $x_2 = 1$ (from $P_2(1, 7)$)
- $x_3 = 3$ (from $P_3(3, 4)$)
- $x_4 = 4$ (from $P_4(4, 3)$)
- $x_5 = 2$ (from $P_5(2, 5)$)
- $x_6 = 5$ (from $P_6(5, 2)$)

The minimum x-coordinate is 1, and the maximum x-coordinate is 7. So, we will create a counting array for the range [1, 7].

Step 2: Counting the Occurrences of Each x-coordinate

We count how many times each x-coordinate appears.

$x = 1$ appears 1 time.
$x = 2$ appears 1 time.
$x = 3$ appears 1 time.
$x = 4$ appears 1 time.
$x = 5$ appears 1 time.
$x = 6$ appears 0 times.
$x = 7$ appears 1 time.

Step 3: Construct the Sorted List

We now create the sorted list of points based on the x-coordinate, using the counts.

Sorted points according to the x-coordinate:

- $P_2(1, 7)$
- $P_5(2, 5)$
- $P_3(3, 4)$

- $P_4(4, 3)$
- $P_6(5, 2)$
- $P_1(7, 1)$

Final Result:

The points sorted according to the x-axis are:

- $P_2(1, 7)$
- $P_5(2, 5)$
- $P_3(3, 4)$
- $P_4(4, 3)$
- $P_6(5, 2)$
- $P_1(7, 1)$

[0.5 Points] What is the complexity using Computing Sort for the above mentioned scenario: O(6),O(13) or O(12)

The time complexity of **Counting Sort** depends on two main factors:

1. **Range of values (k)**: The difference between the minimum and maximum values in the input array.
2. **Number of elements (n)**: The total number of items being sorted.

In your scenario:

- **n = 6** (because there are 6 towers).
- **k = 7** (since the x-coordinates range from 1 to 7).

Complexity Analysis:

- **Step 1**: **Counting Occurrences** - This step involves iterating over the 6 towers, which takes **O(n)** time.
- **Step 2**: **Computing Cumulative Counts** - This step iterates over the counting array, which has a size of **k**. Thus, this step takes **O(k)** time.
- **Step 3**: **Placing the Elements** - This step also involves iterating over the 6 towers and placing them at their correct positions, which takes **O(n)** time.

Thus, the total time complexity for this sorting process is:

$$O(n+k) = O(6+7) = O(13)$$

Since **n = 6** and **k = 7**, the time complexity is **O(n + k)**, which simplifies to **O(13)** in this case. However, **O(n + k)** is generally the most accurate representation for the time complexity of Counting Sort, as the actual complexity will depend on both the number of elements and the range of values.

In summary:

- **The time complexity for this scenario is O(n + k)**, or **O(13)**. If you want to refer to the constant 6 as part of the complexity, it's **O(6 + 7) = O(13)**.
- The notation **O(6)** is not the most accurate way to express complexity since it ignores the dependency on the range of values **k**. Instead, the complexity should be expressed as **O(n + k).**

[0.5 Points] Split the points with line L so that half the points are on each side. Consider the verticalline passing through the middle point (4,3). Plot all the points

To split the points with a vertical line passing through the middle point (4, 3), we will consider the line **L** as a vertical line with the equation **x = 4**. This means that all points with an x-coordinate less than 4 will be on the left of the line, and all points with an x-coordinate greater than 4 will be on the right of the line.

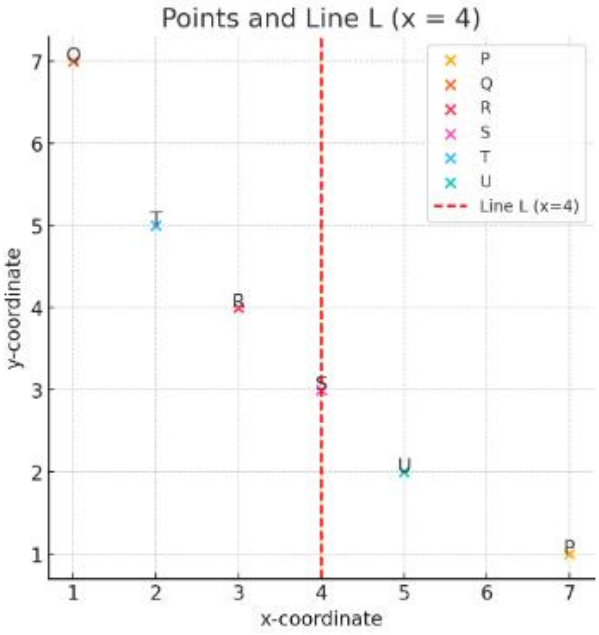
Points to plot:

1. Tower P: (7, 1) — $x > 4$ (to the right of the line)
2. Tower Q: (1, 7) — $x < 4$ (to the left of the line)
3. Tower R: (3, 4) — $x < 4$ (to the left of the line)
4. Tower S: (4, 3) — $x = 4$ (on the line itself)
5. Tower T: (2, 5) — $x < 4$ (to the left of the line)
6. Tower U: (5, 2) — $x > 4$ (to the right of the line)

Step-by-Step:

- **Left of the line (x < 4)**: Towers Q, R, and T
- **On the line (x = 4)**: Tower S
- **Right of the line (x > 4)**: Towers P and U

Let plot these points for you, showing the line **L** (x = 4).



Here is the plot showing the points and the vertical line **L** passing through (4, 3). The line **L** divides the points as follows:

- **Left of the line (x < 4)**: Towers Q, R, and T
- **On the line (x = 4)**: Tower S
- **Right of the line (x > 4)**: Towers P and U

The line **L** is represented by the red dashed vertical line at **x = 4**

[1 Point] Use City Block Distance and Brute Force Approach to compute distance of all points onthe left hand side of the Line L and right hand side of the Line L.

To compute the City Block Distance (also known as Manhattan Distance) between each pair of points on the left-hand side of the vertical line $x = 4$, we'll use the points that lie on the left of this line.

From the previous split, the points on the left-hand side of the line are:

- $P_2(1, 7)$
- $P_3(3, 4)$
- $P_5(2, 5)$

The **City Block Distance** (Manhattan Distance) between two points (x_1, y_1) and (x_2, y_2) is calculated as:

City Block Distance = $|x_1 - x_2| + |y_1 - y_2|$

Compute Distances Between All Pairs of Points on the Left-Hand Side

Distance between $P_2(1, 7)$ **and** $P_3(3, 4)$**:**

Distance = | 1 – 3 | + | 7 – 4 | = 2 + 3 = 5

Distance between $P_2(1, 7)$ **and** $P_5(2, 5)$**:**

Distance = | 1 – 2 | + | 7 – 5 | = 1 + 2 = 3

Distance between $P_3(3, 4)$ **and** $P_5(2, 5)$**:**

Distance = | 3 – 2 | + | 4 – 5 | = 1 + 1 = 2

The distances between all pairs of points on the left-hand side are:

- Distance between $P_2(1, 7)$ and $P_3(3, 4)$: **5**
- Distance between $P_2(1, 7)$ and $P_5(2, 5)$: **3**
- Distance between $P_3(3, 4)$ and $P_5(2, 5)$: **2**

These distances were computed using the brute-force approach and City Block (Manhattan) Distance formula.

To compute the City Block Distance (Manhattan Distance) between each pair of points on the right- hand side of the vertical line $x = 4$, we'll use the points that lie on the right of this line.

From the previous split, the points on the right-hand side of the line are:

- $P_1(7, 1)$
- $P_6(5, 2)$

Formula for City Block Distance

The **City Block Distance** (Manhattan Distance) between two points (x_1, y_1) and (x_2, y_2) is calculated as:

City Block Distance = $|x_1 - x_2| + |y_1 - y_2|$

Compute the Distance Between Points on the Right-Hand Side

Distance between $P_1(7, 1)$ **and** $P_6(5, 2)$**:**

Distance = | 7 – 5 | + | 1 – 2 | = 2 + 1 = 3

The only distance between the points on the right-hand side is:

Distance between $P_1(7, 1)$ and $P_6(5, 2)$: **3**

This distance was computed using the brute-force approach and the City Block (Manhattan) Distance formula. Since there are only two points on the right-hand side, we have just one distance to compute.

[0.25 Point] What is the minimum distance dl on left hand side and dr on right hand side? Calculatethe minimum of d = min(dl,dr)

To find the minimum distance on the left-hand side (d_l) and the right-hand side (d_r), we need to compare the smallest distances within each side, and then determine the minimum of these two values.

Left-Hand Side Distances:

The points on the left-hand side of the line $x = 4$ are:

- $P_2(1, 7)$
- $P_3(3, 4)$
- $P_5(2, 5)$

We already computed the distances between all pairs on the left-hand side:

- Distance between $P_2(1, 7)$ and $P_3(3, 4)$: **5**
- Distance between $P_2(1, 7)$ and $P_5(2, 5)$: **3**
- Distance between $P_3(3, 4)$ and $P_5(2, 5)$: **2**

The minimum distance on the left-hand side, d_l, is **2**.

Right-Hand Side Distance:

The points on the right-hand side of the line $x = 4$ are:

- $P_1(7, 1)$

  $P_6(5, 2)$

Thus, the minimum distance on the right-hand side, d_r, is **3**.

Calculate the Minimum of d_l and d_r

We need to find the minimum distance between the two sides:

d = min(d_l, d_r) = min(2, 3) = 2

The minimum distance d between the closest points on the left and right sides of the line is **2**.

[0.5 Point] Find points close (closer than d) to the line passing through the middle point

To find the points that are closer than $d = 2$ to the vertical line passing through the middle point $(4, 3)$, we need to compute the **City Block Distance** from each point to the line $x = 4$, and identify which points have a distance less than 2.

The **City Block Distance** from a point (x_i, y_i) to the vertical line $x = 4$ is simply the absolute difference between the x-coordinate of the point and 4:

Distance to the line = $|x_i - 4|$

Step 1: Calculate the Distance of Each Point from the Line $x = 4$

For each point, we calculate the City Block Distance to the line $x = 4$:

For $P_2(1, 7)$:

$$\text{Distance} = |1 - 4| = 3$$

For $P_3(3, 4)$:

$$\text{Distance} = |3 - 4| = 1$$

For $P_5(2, 5)$:

$$\text{Distance} = |2 - 4| = 2$$

For $P_1(7, 1)$:

$$\text{Distance} = |7 - 4| = 3$$

For $P_6(5, 2)$:

$$\text{Distance} = |5 - 4| = 1$$

For $P_4(4, 3)$:

$$\text{Distance} = |4 - 4| = 0$$

Step 2: Identify Points Closer Than $d = 2$

We now check which points have a distance from the line less than 2:

- $P_2(1, 7)$: Distance = 3 (not closer than 2)
- $P_3(3, 4)$: Distance = 1 (closer than 2)
- $P_5(2, 5)$: Distance = 2 (not closer than 2) • $P_1(7, 1)$: Distance = 3 (not closer than 2) • $P_6(5, 2)$: Distance = 1 (closer than 2)
- $P_4(4, 3)$: Distance = 0 (closer than 2)

Step 3: Result

The points that are closer than 2 units to the vertical line passing through $(4, 3)$ are:

- $P_3(3, 4)$
- $P_6(5, 2)$
- $P_4(4, 3)$

These are the points whose City Block Distance from the line $x = 4$ is less than 2.

[0.25 Point] Sort points obtained in (f) using y-coordinate. No need to apply any algorithm. Just manually sort these

The points obtained in step (f) that are closer than 2 units to the vertical line $x = 4$ are:

- $P_3(3, 4)$
- $P_6(5, 2)$
- $P_4(4, 3)$

Sorting Points by the y-coordinate:

1. $P_6(5, 2)$: y-coordinate = 2
2. $P_4(4, 3)$: y-coordinate = 3
3. $P_3(3, 4)$: y-coordinate = 4

Sorted Points (by y-coordinate):

$P_6(5, 2)$
$P_4(4, 3)$
$P_3(3, 4)$

These are the points sorted in increasing order of their y-coordinates.

[0.5 Point] Find pair of points with difference between y coordinates is smaller than d

To find the pair of points where the difference in their **y-coordinates** is smaller than $d = 2$, we need to check the difference between the y-coordinates of each pair of points from the sorted list:

-
-
-

The points obtained in step (f) are:

- $P_6(5, 2)$
- $P_4(4, 3)$
- $P_3(3, 4)$

Calculate the Difference in y-coordinates Between Each Pair

Difference between $\mathrm{P}_6(5, 2)$ **and** $\mathrm{P}_4(4, 3)$**:**

Difference in y-coordinates = | 2 – 3 | = 1

Difference between $\mathrm{P}_6(5, 2)$ **and** $\mathrm{P}_3(3, 4)$**:**

Difference in y-coordinates = | 2 – 4 | = 2

Difference between $\mathrm{P}_4(4, 3)$ **and** $\mathrm{P}_3(3, 4)$**:**

Difference in y-coordinates = | 3 – 4 | = 1

Identify Pairs with y-coordinate Difference Smaller Than $d = 2$

- The difference between $P_6(5, 2)$ and $P_4(4, 3)$ is **1**, which is smaller than 2.
- The difference between $P_4(4, 3)$ and $P_3(3, 4)$ is **1**, which is smaller than 2.
- The difference between $P_6(5, 2)$ and $P_3(3, 4)$ is **2**, which is **not smaller** than 2.

Result

The pairs of points where the difference in their y-coordinates is smaller than 2 are:

$(P_6(5, 2), P_4(4, 3))$
$(P_4(4, 3), P_3(3, 4))$

[0.25 Point] Use Brute Force to compute distances b/w all points and assign minimum as d'

To compute the distances between all points using the **Brute Force** approach and assign the minimum distance as d', we will compute the **City Block Distance** (Manhattan Distance) between every pair of points from the set:

- $P_6(5, 2)$
- $P_4(4, 3)$
- $P_3(3, 4)$

Compute the City Block Distances Between All Pairs of Points

We will use the formula for **City Block Distance**:

City Block Distance = $|x_1 - x_2| + |y_1 - y_2|$

Distance between $P_6(5, 2)$ **and** $P_4(4, 3)$**:**

Distance = $|5 - 4| + |2 - 3| = 1 + 1 = 2$

Distance between $P_6(5, 2)$ **and** $P_3(3, 4)$**:**

Distance = $|5 - 3| + |2 - 4| = 2 + 2 = 4$

Distance between $P_4(4, 3)$ **and** $P_3(3, 4)$**:**

Distance = $|4 - 3| + |3 - 4| = 1 + 1 = 2$

Step 2: Find the Minimum Distance

The distances between all pairs of points are:

- Distance between $P_6(5, 2)$ and $P_4(4, 3)$: **2**
- Distance between $P_6(5, 2)$ and $P_3(3, 4)$: **4**
- Distance between $P_4(4, 3)$ and $P_3(3, 4)$: **2**

The **minimum distance** d' is the smallest of these distances:

$d' = \min(2, 4, 2) = 2$ The minimum distance d' between any pair of points is **2**.

[0.25 Point] Which one is smaller d or d' and this is your final shorted pair using divide and conquer

We have two distances to compare:

- $d = 2$ (the minimum distance between points on opposite sides of the line $x = 4$).
- $d' = 2$ (the minimum distance computed using the brute force method between all points on the left and right sides).

Comparison:

Since both d and d' are equal to **2**, neither is smaller than the other.

Final Sorted Pair Using Divide and Conquer:

Now, we will use the **Divide and Conquer** approach to find the closest pair of points. In this case, since both d and d' are equal to 2, the closest pair can be found by considering the points that are the
closest to each other.

From our earlier results, the closest pair with a distance of 2 are:

- $(P_6(5, 2), P_4(4, 3))$
- $(P_4(4, 3), P_3(3, 4))$

Both pairs have a distance of 2, which is the minimum distance computed.

Thus, using the **Divide and Conquer** approach, **both pairs** (P_6, P_4) and (P_4, P_3) can be considered the closest pair with the minimum distance of 2.