# ALGORITHM ANALYSIS AND DESIGN

PROJECT

Roll no:2020101097
Yeduru Rupasree

# INDEX

# SORTING


**SELECTION SORT:**

Suppose there are 5 children in a class standing in a line and our task is to sort them in increasing order of their heights.

We first select the first person in the line and compare him with the second person.If the second person is shorter than first person, exchange their positions and continue comparing first person with third,fourth and fifth persons.In this process we will find the shortest person in the first position.

So our next task is to sort the next four persons.We continue a similar process selecting second,third,fourth positions starting comparison from third,fourth and fifth positions respectively.

**Pseudocode:**
*Input:unsorted array A[0],.....A[n-1]*

*Algorithm:*
*For i=0…..n-2:*

*For j=i+1...n-1:*

*-ascending*
*{if(A[i]>A[j])*
*swap(A[i],A[j])}*

*-descending*
*{if(A[i]<A[j])*
*swap(A[i],A[j])}*

*Output: sorted array*

**BUBBLE SORT:**

In this sorting,we make larger or smaller elements to bubble towards last.

If we want elements in ascending order,we first move the largest element to the last.To do this starting from the first element we compare adjacent elements and swap them if ith element is larger than the i+1th element.From this step we will get the largest element at nth position.For second largest element we will repeat this process from first position to n-1th position.We repeat this processes till we get n-1th largest element at second position.
For descending order swap if ith element is smaller than the i+1th element.

**Pseudocode:**
*Input:unsorted array A[0],.....A[n-1]*

*Algorithm:*
*For i=0…..n-2:*

*For j=0….(n-i-1)*
*-ascending*
*{if(A[j]>A[j+1])*
*swap(A[i],A[j])}*
*-descending*
*{if(A[j]<A[j+1])*
*swap(A[i],A[j])}*
*Output: sorted array*

**INSERTION SORT:**

*Have you ever played cards??*
*Insertion sort is just similar to arranging cards.*

We place sorted and unsorted parts separately,pick one from the unsorted part ,and try to arrange it along with the sorted part by comparing elements in the sorted array.

*Input: unsorted array A[0].....,A[n-1]*

*Algorithm:*
*For i=1……...n:*

```
key = arr[i];
j = i - 1;
while  (j >= 0 && arr[j] > key) :
    arr[j + 1] = arr[j];
    j = j - 1;
arr[j + 1] = key;
```

*Output: sorted array*

**COUNTING SORT:**

The Counting Sort Algorithm is a fast sorting algorithm that can be used to sort elements within a specific range that contains several elements that are repeated. This method of sorting is based on the count of each element and hence *counting sort!!*

*Input: unsorted array A[0].....,A[n-1] with range 0 to k*

*Algorithm:*

*Initialize an array count of size k and store the frequency of every distinct element.*

*Update the count array so that element at each index i as follows:*

$$count[i] = \sum count[u] \quad \text{where } 0 <= u <= i$$

*For i=0……...n-1:*

*j=A[i]*

*B[l] =j such that l = (count[j]-1).*

*count[t] --*

*Output: Sorted array B[0].....B[n-1]*

***Disadvantages of Counting Sort:***

It is not suitable for sorting data sets with large range

It is not suitable for sorting string values

# GREEDY ALGORITHMS

## HUFFMAN CODE:

***Our goal is to figure out the best cost-effective approach to write a large string in binary.***

 ***What's the problem with using the same number of bits for all characters???***

Suppose we are using the same number of bits to represent each character, we'll need a lot of bits. Instead, we can lower the number of bits by using small length encoding for high-frequency characters and bigger length encoding for low-frequency characters.

***Will choosing random strings based on frequencies suffice ?***

We must be cautious while selecting this codeword.

For example, if **A-1 B-0 C-10** are used ,consider the encoding 100, it represents two different strings, **ABB and**

**CB.** As a result, we must choose our code words in such a way that they cannot be used as a prefix to another code word.

***How to fix it??***

The easiest approach to represent these codewords is to represent it in the form of a binary tree, where each leaf node represents a code word and travelling to the right indicates 1 and travelling to the left represents 0 going from the root node to the leaf node.Because a leafnode can't be a predecessor node of any other leafnode in its path, this representation works.

***Efficient way:***

From this representation, we can acquire various length encodings with prefix free properties, but our goal is to reduce the overall length of the encoding.

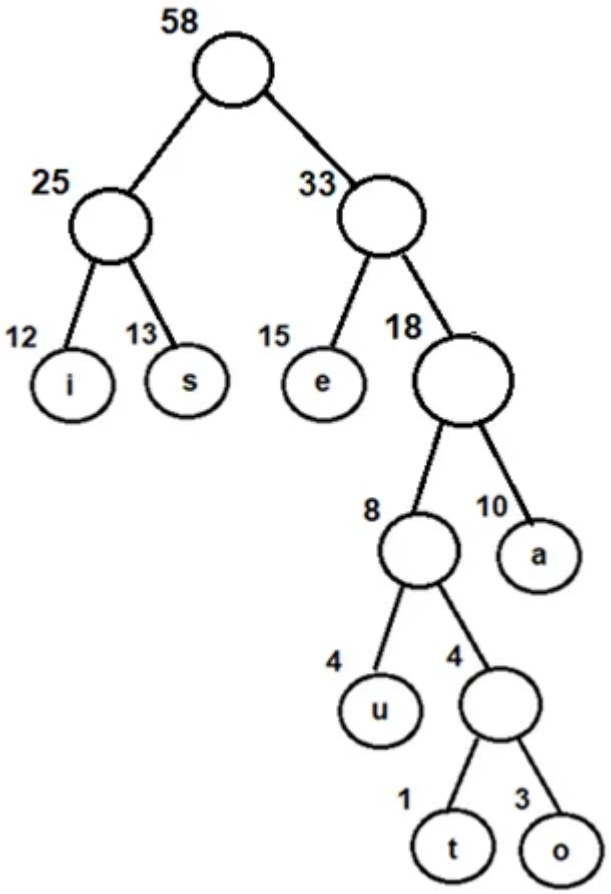
If f1,f2,...fn denotes the frequency of characters, then the total cost of encoding equals the cost of the tree.

$$\text{Cost of the tree} = \sum_{i=1}^{n} f_i . \text{depth of node}$$

We follow a greedy approach here, choosing low-frequency characters to be represented as leaves with more depth and high-frequency characters to be portrayed as leaves with less depth. As a node is traversed only while traversing its descendant nodes, the frequency of an internal node would be the total of the frequencies of its descendants.

Because each node in the tree, except the root node, represents a bit based on whether it is a right or left child, the total length of encoding would equal the sum of frequencies of all nodes.

**Example:**




---

**ACTIVITY SELECTION:**

*Suppose you have to do several activities and each of your activities has certain start and finish times and you have to complete a maximum number of activities without overlapping,what will you do?*

Once we identify the first activity that provides the best solution, we can apply the same methodology to the remaining activities that are compatible with it.

***How to choose the first activity to be completed?***

Use a greedy approach, select the activity that has the least finish time!!

Sort the activities based on their finish times
Select the first activity from the sorted array.
From remaining activities,select the next activity ,if the start time of this activity is longer than the finish time of the previously selected activity.

---

**SET COVER:**

*Suppose there are four persons A with skills {a,b,c},B with skills {b,c} , C with skills {a,b} , D with skills {c,d} and we have to select minimum number of persons so that persons with all skills {a,b,c,d } should be present ,we will select A and D as they cover all skills a,b,c,d and removing any one from them would not suffice.*

Set cover problem is similar to it!!

***Problem:***

Given a set of elements B and collection of sets S1,S2,....Sn our goal is to find collection sets Si whose union is B.The optimum solution covers minimum no of sets Si whose union is B.

***How to select optimum cover??***

Let's try a greedy approach here.Select the set with the greatest number of uncovered elements each time, then repeat the process for the remaining uncovered components.

***Will it always work?***

The greedy approach appears to work well, but it does not always give the optimal solution. While it can produce a collection of sets that gives union , it may sometimes cover more sets than the minimum number of sets possible.

Consider the following examples:

*1.B={1,2,3} ,S1={1,2} ,S2={1,3} ,S3={1}*

The greedy technique chooses S1 first since it has the most exposed components and then S2 such that S1 U S2=B, with cost=2. It is the best answer because no collection of sets with a cost of 1 yields B as the union. In this case, the greedy strategy gave the optimal solution.

*2.B={a,b,c,d,e,f}, S1 = {a, b, c, d },S2 = {a, b, f} , S3 ={ c, d, e}*

The greedy technique chooses S1 first since it has the most exposed components, then S2 and finally S3, resulting in S1 U S2 U S3 as the solution with cost 3. However, S2 U S3 is also possible at cost 2, which is the best solution. In this case, a greedy approach does not yield the best result.

*When it comes to set cover, a greedy approach isn't always the optimal solution.*

# DYNAMIC PROGRAMMING

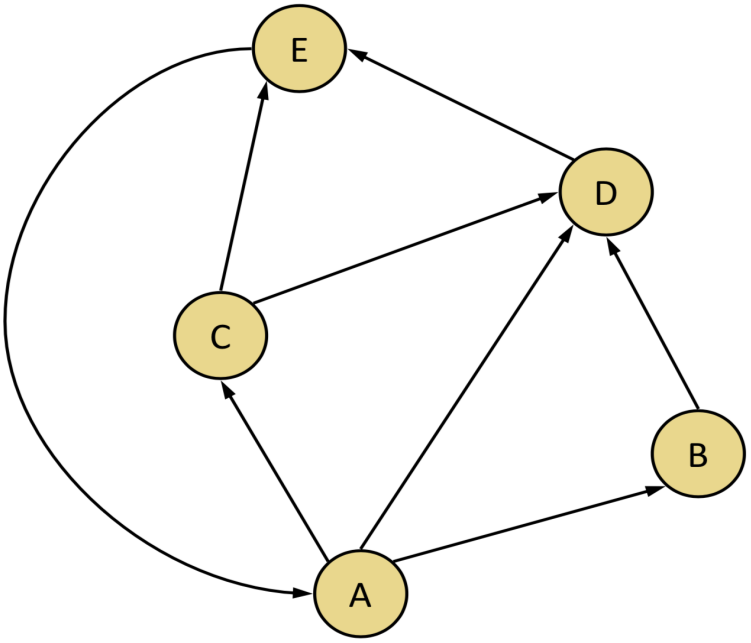
## SHORTEST PATH IN DAG:

*Our goal is to find the shortest path between two points in a directed acyclic graph.*

Because each subproblem is predicated on smaller subproblems, we employ a dynamic programming method here.

We need to identify shortest paths for all nodes N if there is a directed edge from N to S to determine the shortest path between an origin O and a point S.

We will be able to identify the shortest path from S using these shortest paths and the weight of edges related to S to get the one with the smallest length among those paths.




The shortest path between A and E, for example, would be the minimum of {shortest path(A,C)+CE, shortest path(A,D))+DE }

*shortest path(V)=min{shortest path(U)+length(U,V)} if(U,V) belongs to E.*

We must begin with the smallest problem because each subproblem is predicated on a smaller subproblem. We begin with origin O and a shortest path length of 0, assuming that all other vertices have infinity shortest path lengths.

Then we identify the shortest paths for all vertices that are related to it, and we repeat the process for vertices that are connected to them.

***Algorithm:***

*initialize all dist(·) values to* $\infty$

*dist(O) = 0*

*for each* $v \in V\{O\}$*, in linearized order:*

$dist(v) = \min_{(u,v)\in E} \{dist(u) + l(u, v)\}$

Similarly to find the longest path in dag we find max every time instead of min.

*longest path(V)=max{longest path(U)+length(U,V)} if(U,V) belongs to E.*

---

***What is a subsequence??***

*A subsequence is formed by removing some elements from the original sequence without changing their order.*

**LONGEST COMMON SUBSEQUENCE:**

*A sequence Z is said to be a common subsequence of X and Y if it is a subsequence of both X and Y given two sequences X and Y.*

For example:
If X=A,C,B,D,E,G,C,E,D,B,G, and Y=B,E,G,C,F,E,U,B,K, then

Z={B,E,E}
is then considered a common subsequence of X and Y. Because Z only has three elements while the common subsequence {B,E,E,B} has four elements, this would not be the longest common subsequence.
{ B,E,G,C,E,B} is the LCS of X and Y.

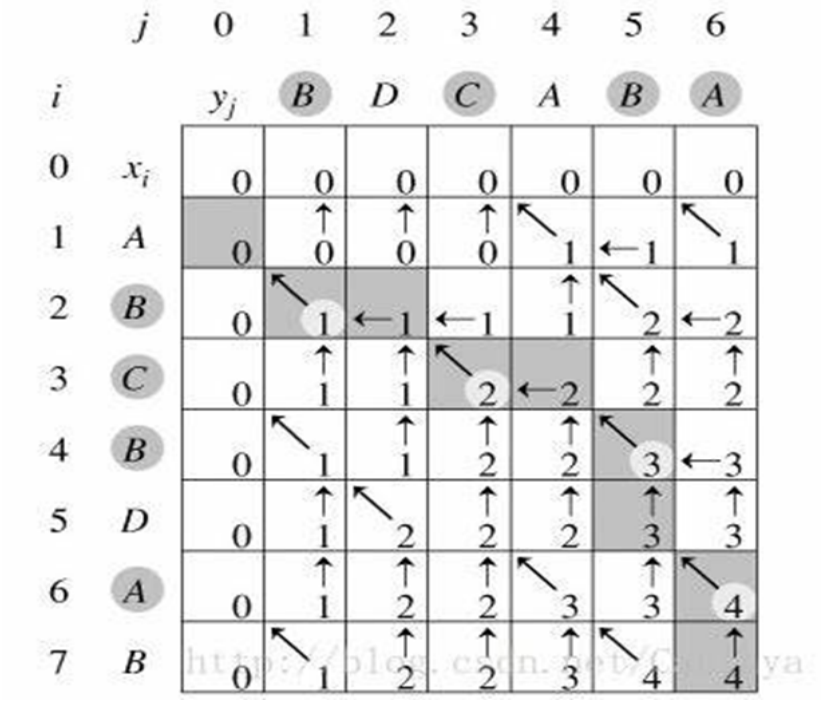
In dynamic programming we construct a 2D table L[m+1][n+1].The value of L[m][n] contains the length of LCS.

***Algorithm:***

$$LCS[i][j] = \begin{cases} 0 \text{ if } i=0 \text{ or } j=0 \\ LCS[i-1][j-1]+1 \text{ if } A[i-1]==B[j-1] \\ \max\{LCS[i][j-1], LCS[i-1][j]\} \text{ if } A[i-1] \neq B[j-1] \end{cases}$$

Here,the least common subsequence LCS[i][j] depend on the sub problems LCS[i][j-1],LCS[i-1][j],LCS[i-1][j-1] based on condition.

*Example:*
To find LCS of ABCBDAB and BDCABA:




Fill the table based on algorithm ,follow the arrows from last,for each diagonal arrow there is a member of LCS

---

## LONGEST INCREASING SUBSEQUENCE:

*The goal of the Longest Increasing subsequence problem is to identify a subsequence of a given sequence that has all of its parts ordered from lowest to highest and is as long as possible.*

Our subsequence, on the other hand, need not begin with the first element. We need to discover the longest subsequence out of all feasible longest increasing subsequences starting from various elements.

The longest increasing subsequence ending at an element aj is derived by adding it to the longest increasing subsequence terminating at all ai's, where ai<aj & i<j. L(j) is the length of the longest increasing subsequence that ends at the jth element.

***L(j)=max {1+ L(i) }*** with i's satisfying above conditions.

We start with the first element of the sequence and find the length of the LIS for all elements since subproblems can be solved once smaller subproblems are solved.

## HOW IS LIS RELATED TO LCS ?

*The longest increasing subsequence problem is closely related to the longest common subsequence problem, the longest increasing subsequence of a sequence S is the longest common subsequence of S and T, where T is the result of sorting S.*

## HOW IS LIS RELATED TO LONGEST PATH IN DAG ?

*The longest subsequence problem can be converted into dag with edge from one element to another element if they can be a part of an increasing subsequence.*

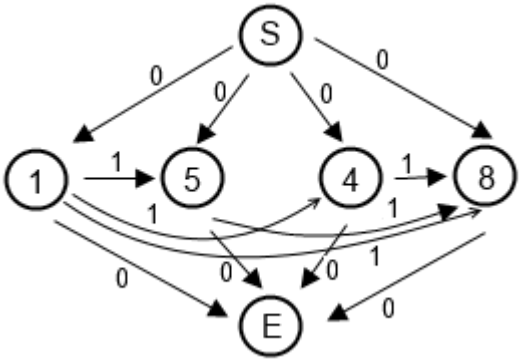
*There will be a directed edge from ai,aj if i<j and ai<aj each with edge weight 1.*

*Add a node S that has outgoing edges of weight 0 to all the number vertices.*

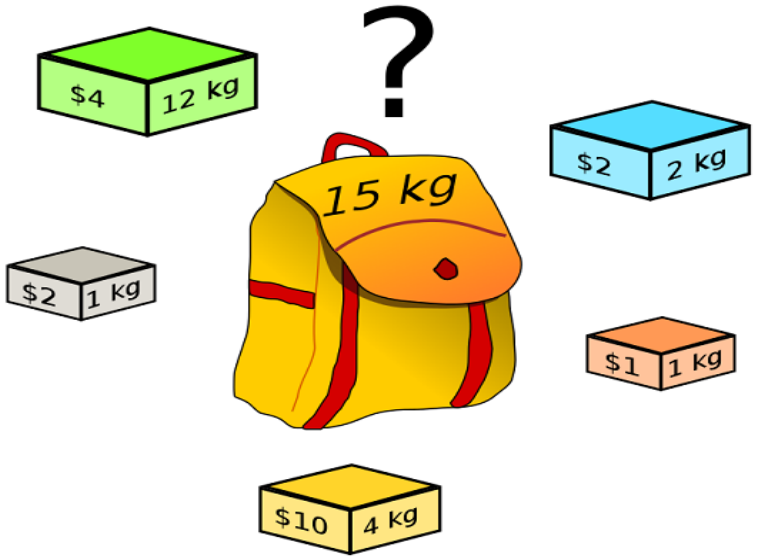
*Add a node E that has incoming edges of weight 0 from all the numbers vertices.*

*Longest Increasing Subsequence would be along the longest path in dag created in the above way.*

***Example:***



**KNAPSACK PROBLEM:**




*Our aim is to identify the most valuable combination of goods that can fit into a knapsack of capacity W, given n items with weights w1,w2...wn and values v1,v2,v3...vn.*

***a.Repetition is allowed:***

K(w), where K(w) represents the cost of goods with total weight w, is our subproblem. When you remove an item with value vi from it, you'll get a knapsack with weight w-wi and a cost of K.(w-wi). To get weight w and value K(w-wi)+vi,we need to add an item I with weight wi and value vi to a knapsack with weight (w-wi).

To find K(w) we need to find smaller subproblems K(w-wi) for i=1....n.

As a knapsack with a weight of zero has a value of zero, K (0). To find K(W) from smaller subproblems, we start with K(0) and iterate from 1....W.

***Algorithm:***

*K (0) = 0*

*for w = 1 to W:*

*K (w) = max{K (w − wi) + vi : wi ≤ w;}i=1.....n*

*return K (W)*

***b.Repetition is not allowed:***

We can't use K(w) as a subproblem since we don't know whether K(w-wi) contains wi or not, as we need it because there isn't any repetition.

Here, we'll use K(w,j) as a subproblem, where K(w,j) indicates the most valuable weight knapsack that can be created with elements 1.....j.

K(w,j) can be computed in two ways: with the jth item considered and without the jth item considered.

Considering that the jth item costs K(w-wj,j-1)+vi (wj<=w)

Without taking into account the jth item's cost K(w,j-1)

We need to discover the more valuable knapsack from these two scenarios.

*K(w,j)=max {K(w-wj,j-1)+vi, K(w,j-1)} if wj<=w*

*Otherwise, K(w,j)=K(w,j-1)*

since wj>w cannot be made into a knapsack by adding the jth member.

***Algorithm:***

*Initialize all K (0, j) = 0 and all K (w, 0) = 0*

*for j = 1 to n:*

*for w = 1 to W:*

*if wj > w: K (w, j) = K (w, j − 1)*

*else: K (w, j) = max{K (w, j − 1), K (w − wj, j − 1) + vj}*

*return K (W, n)*

---

**SHORTEST RELIABLE PATHS:**

*Given a graph G , nodes s and t in G, we need to find the shortest path from s to t that can be made using at most k edges.*

We can't use just d(v) as a subproblem since we also need to account for the number of edges in the path as well. Here, we'll use d(v,i) as a subproblem, with d(v,i) representing the shortest path from s to v with at most i edges.

To discover the shortest path between s and v, we must first find the shortest paths between s and u( for all u ,(u,v) belongs to E) and to make shortest path between s and v with atmost i edges shortest path between s and u must be with atmost i-1 edges So, for all u connected to v, we get d(u,i-1)+l(u,v) and find the minimum among them.

As a result,

***d(v,i)= {min d(u,i-1)+l(u,v)} where (u,v) is part of E.***

# DIVIDE AND CONQUER

***What is divide and conquer approach ?***

1. *Divide: Break the given problem into subproblems of the same type.*
2. *Conquer: Recursively solve these subproblems.*
3. *Combine: Appropriately combine the answers.*

**BINARY SEARCH:**

Searching for an element linearly in an array would take O(n) time complexity

Binary search is useful for arrays which are already sorted.

We follow a divide and conquer approach here.

Divide array into two sub parts, compare the mid value with given key and search in a subpart recursively based on comparison.

***Pseudocode:***

*Input: Sorted array A[n],key,left and right ends*

```
if (r >= l) {
    int mid = l + (r - l) / 2;

    // If the element is present at the middle
    // itself
    if (arr[mid] == x)
        return mid;

    // If element is smaller than mid, then
```

```
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
```

*Output: index of element if present else -1*

**MERGE SORT:**

In merge sort each array is divided into two parts,these parts are sorted recursively and merged into a sorted array.

***Pseudo Code:***

*Input:arr[N],beg,end*

*MERGE_SORT(arr, beg,end)*

*Algorithm:*

*if beg < end*

*mid = (beg + end)/2*

*MERGE_SORT(arr, beg, mid)*

*MERGE_SORT(arr, mid + 1, end)*

*MERGE (arr, beg, mid, end)*

*Output:Sorted array*

**MAXIMUM SUBARRAY SUM:**

*A[n]:*


i j

*Max subarray sum =max{ X[i]+X[i+1].....X[j]} for all 0<=i<=j<=n*

For example:

Consider A[]={-4,5,7,-6,10,-15,3 } ,the sub array {5,7,-6,10} has the maximum sum of all subarrays.

A naive approach starting from each element and checking all possible subarray sums would take O(n^3)

Let's do divide and conquer approach here,

For an array the subarray has 3 possibilities

It can be in left part of array,right part of array or including middle element

We divide them in such a way and find the maximum possible sum among them.

We find max subarray in the left part and right part following the same approach recursively.

To find max subarray crossing mid point we find max sub arrays A[i…..mid],A[mid+1,....j] and combine them.

After calculating these 3 subarrays we take the maximum of them.

**CLOSEST PAIR OF POINTS** :

*Given a set of n points in a 2D plane,our goal is to find the pair of points which are closest of all pairs.*

Brute force approach finding distance between every pair of points takes O(n) time

Lets approach divide and conquer approach

Divide the array of points into two parts to recursively find the closest pair in each part.

*But there might be a case where one point from each side makes the closest pair!!*

To consider that case,

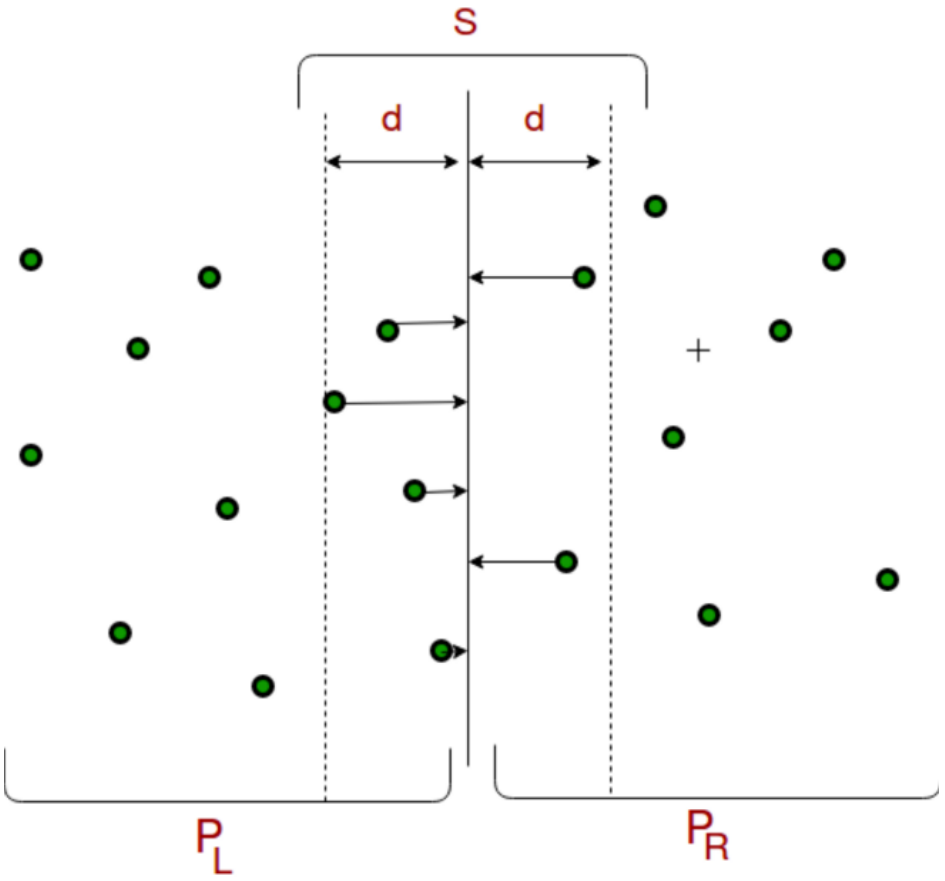
Compute the minimum distances in the previous cases:

$(p_1, q_1)$ = *ClosestPair(Lx, Ly)*

$(p_2, q_2)$ = *ClosestPair(Rx, Ry)*

*Let* $d = min\{d(p_1, q_1), d(p_2, q_2)\}$

Consider the vertical line that passes through P[n/2], and find any places with x coordinates that are closer to the centre vertical line than d. Create an array Sy of all such points, sorted by y-coordinate, by reading all points in Py and validating their x-coordinates.




Iterate through all the points in Sy with its 1st to 7th adjacent points and update the best closest pair.

Let (p3,q3) represents Closest Split Pair(Px,Py,d)

Find minimum of (p1,q1),(p2,q2),(p3,q3)