1.

(a). What order does Kruskal’s algorithm add edges to the minimum spanning tree?

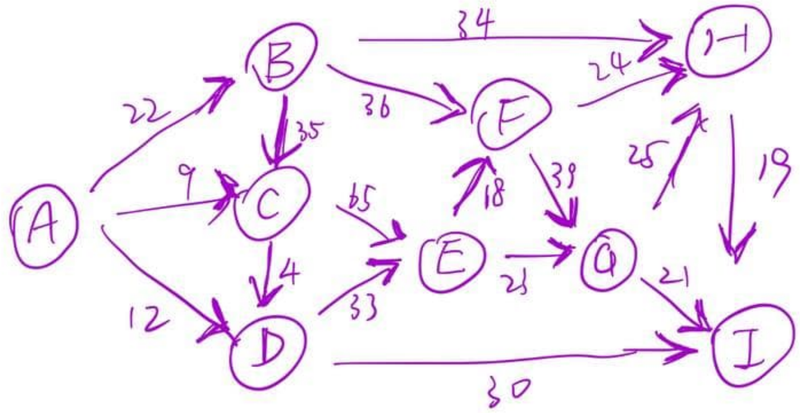
Ans: CD -> AC -> EF -> HI -> GI -> AB -> EG -> DI

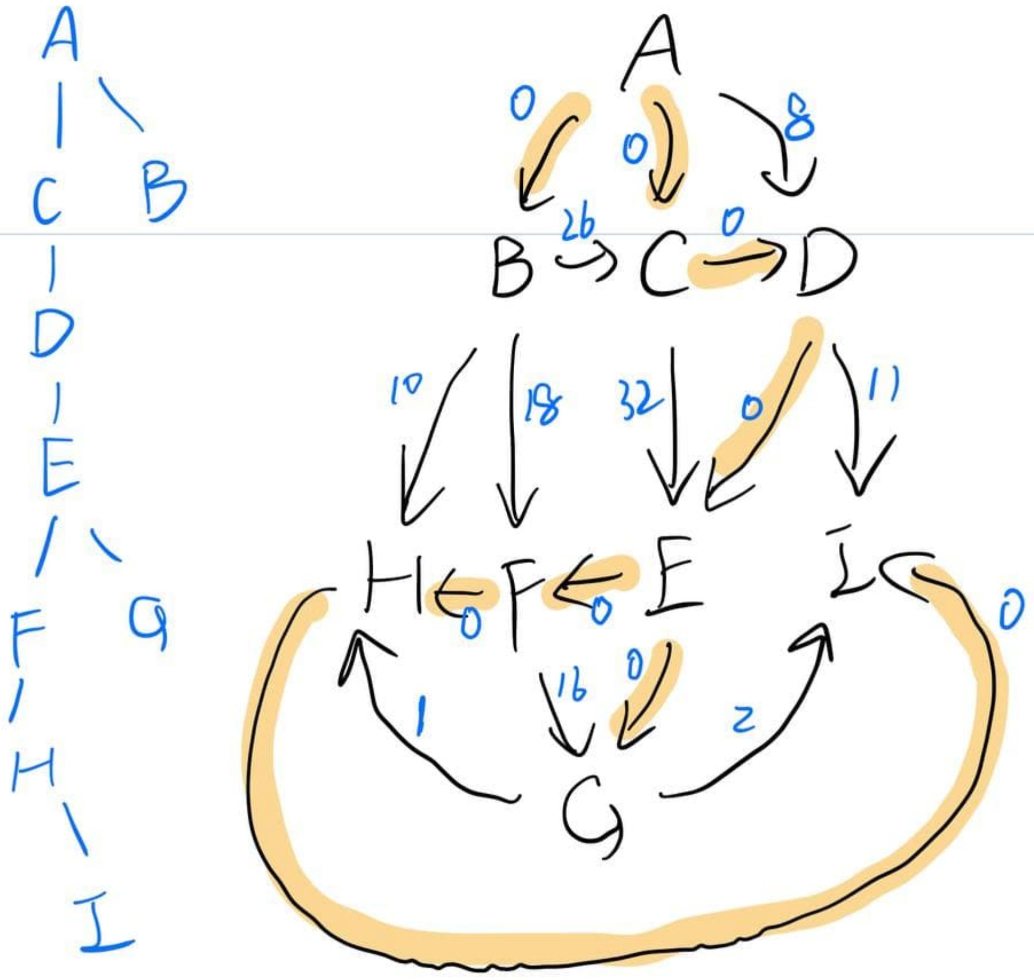
(b). What order does Prim’s algorithm add edges to the minimum spanning tree

Ans: AC -> CD -> AB -> DI -> HI -> GI -> EG-> EF

(c). Use the minimum cost arborescence algorithm to find the minimum cost arborescence rooted at A.

Ans: minCost = AC + AB + CD +DE +EG + EF + FH + HI = 9 + 22 + 4 + 33 + 23 + 18 + 24 + 19 = 152.





2.

(a). $\Theta(n^{\log_2 5}) \approx \Theta(n^{2.321})$
(b). $\Theta(n)$
(c). $\Theta(n^3 logn)$

3.

In order to find two closest pairs of points, we can set return value of out divide & conquer function to be a array which contains of these two points. array[0] is the most closest pair of points and array[1] is the next closest points.
The idea of divide & conquer is very same.

Ans:

General Idea:

In this situation, my two dividing parts(left and right) will return two arrays which each part represent the 1st,2nd closest pair of points in each side. Hence, I am gonna take the 1st, 2nd closest distance to form my new array among these four distances

```
int[] left = findTwoClosesetPairs(X, low, mid);
int[] right = findTwoClosesetPairs(X, mid + 1, high);
int[] newArr = new int[2];
newArr[0] = {1st closest <=> Math.min(left[0], right[0])}
newArr[1] = (2nd closest Math.min(left[0], right[0], left[1], right[1]) without newArr[0]  )
```

Edge cases:

1. when the size the input arr is 2, I will return [d1, MAX_VALUE]
2. when the size the input arr is 3, I will return [d1, d2] which d1 is 1st smallest, d2 is 2nd smallest.

Combining part, instead of use the 1st closest distance from both sides to find possible smaller distance in the middle area(mid-d, mid +d),**We use 2nd closest distance as our d(newArr[1]) to find two closest pairs of points inside this area(mid - newArr[1], mid + newArr[1])**

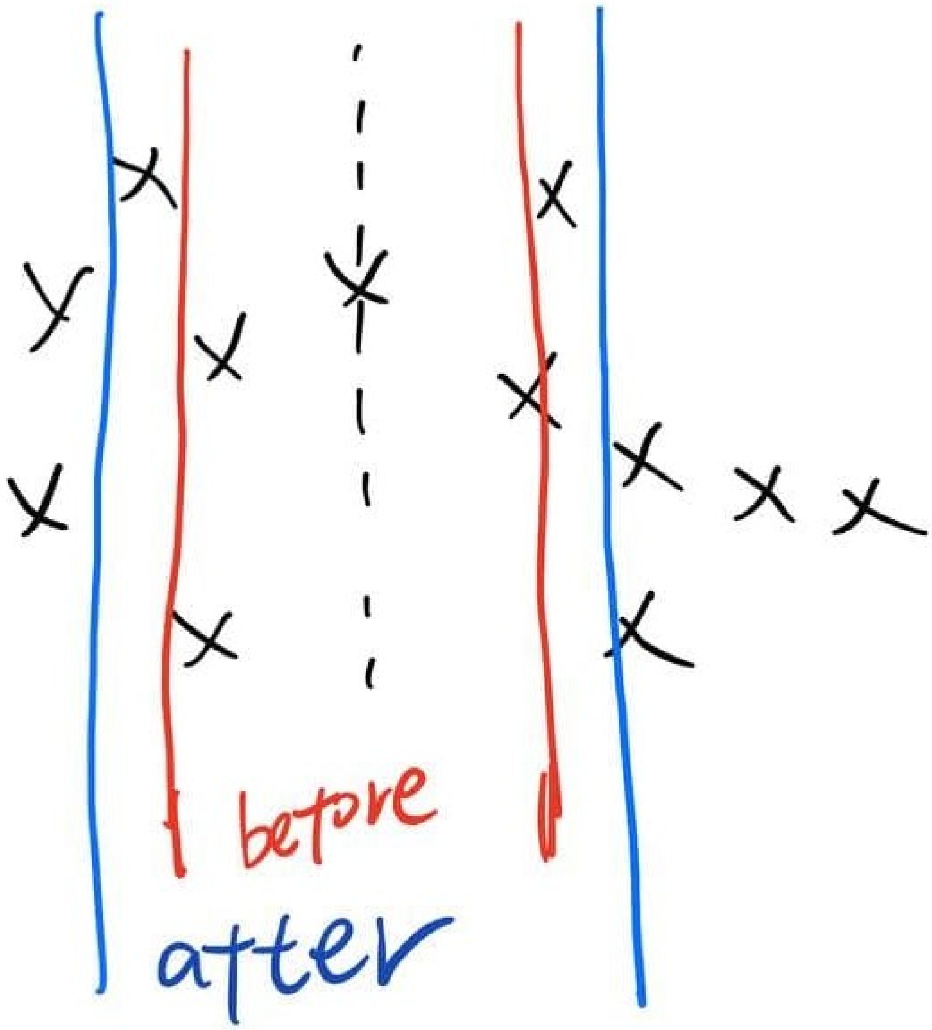
```
possible situation when we calculate distance in the middle area:

1. these are no smaller distance than our 2nd closest(newArr[1]) then:
-> we don't make changes to our newArr

2. these exist d' satisfy  newArr[0] <= d' < newArr[1]
-> so we replace it, newArr[1] = d'

3. these exist d' satisfy  d' < newArr[0]
-> we replace both, newArr[1] = newArr[0] and newArr[0] = d'
```

At Last, the newArr is our answer.

Proof: why we can always find two closest pairs?

The idea of find smallest distance by using middle area(mid - d, mid + d) is proved before.

The different thing is that although I choose the larger distance as my d, my smallest distance that I calculate by the 1st smallest distance before (if exist) **will still be in that range because I increase the area of my original middle area.**



4.

(a) $OPT(i) = max\{OPT(i-1) + S_i, OPT(i-2) + l_{i-1}\}$

OPT(i) means the maximum profit I can get by the end of day i.

(b)
Ans:

- I will construct a one dimensional array called dp with a length of (n + 1)
- dp[i] means maximum profit by the end of day i.
- s_i and l_i means the profit of taking a short job or a long job on day i
- Firstly, intialize my dp array with dp[0] = 0;
- then use the formula above to calculate the current max profit from i = 1 to n which only requires
- when i = 1, i - 2 < 0 so we can set that part to 0 because on day 1 if we take long job we won't gain profit until its next day which is day 2.

(c)
Ans:

- Time Complexity => $\Theta(n)$ since we only need 2n time. (Getting two values and comparing only take constant time for each iteration)
- Space Complexity => $O(n)$ since we only use a array with length of n+1

5.

(a)

- $OPT(i) = max\{OPT(i), OPT(j) + OPT(i-j)\}$ i <- 1 to n/2, j <- 0, i - 1
- $if \quad j == 0, \quad OPT(i-j) = OPT(i) = p_i$

(b)

- Firstly, I construct a one dimension array called dp with a length of n + 1;
- inintialize dp array: dp[0] = 0;
- iterate i from 1 to n, use formula in (a) to get maximun profit.
- in which j <- 1 to n/2 because dp[1] + dp[3] <=> dp[3] + dp[1], we avoid duplicate situation.

(c)
Ans:

- Time Complexity: $\Theta(n^2)$
- Space Complexity: $O(n)$