Manging 200 animals. Sensors. minimize the #sensors

Let p be the topmost point of the line

```
S = Φ
while (the entire line is not completely covered)
{ let s be the sensor that covers p and extends as
     far south as possible.
     S = S ∪ {s}
     p = southern end of s
} return S
```

runtime : O(n)

<u>Correctness</u>: suppose there is an optimal solution O that does not make the selection made by our algorithm. Then the sensor it uses to cover the northmost point does not extend as far south as our choice, so we can swap out their choice & swap in our choice to obtain a new optimal solution O'. Inductively repeat for all sensors in O that we did not choose

Interval. set of P stabs n interval

→ First we sort $I_1, I_2, I_3 .. I_n$ according to its right endpts

```
for (i=0 ; i<I.size(); i++)
{ if cover in I[i]:
        continue;
   else:
        Result.push (I[i].endpoint);
        cover = I[i].endpoint;
} return Result;
```

$$\boxed{\begin{array}{l}\text{Initialization}\\ \text{cover} = -\infty\\ \text{Result} = \{\ \}\end{array}}$$

<u>Why correct</u>: The algo can be wrong if I pick more than one point for an interval. I'm not doing it Again, if we miss any interval, it can be wrong. But I am looping through all the interval. It's not possible Moreover, I am picking always the right most point of an interval (If necessary). This approach is covering the highest possible intervals as they are sorted.

<u>complexity</u>: O(n) if we consider the intervals to be sorted
O(nlogn) if sorting needs to be done.

Time slot, minimum # of tour guide assigned. 5 time slots

→ Let time [ ] be the time slot booked.
We will assign the first booked slot to the guide, then we jump '+5' slots to the 7th slot (guide stays for 5 time slots) then we check if the slot is booked or not. If booked, assign new guide and repeat process. If not booked, move to next slot until we find booked slot until slots completed.

<u>Correctness</u>: Suppose, there is an optimal solution which have a different selection methods, then maximum number of slots assigned would not be enforced as it does in my method. If so, then we can replace that solution with the above solution to get new optimal solution

activity interval, minimum resources required.

→ start by sorting the intervals in non-decreasing order according to it's right endpoint.

```
Initialize cover = a[0].endpoint;
Result = 0;
max = 0
for (a = 0 ; a<a.size(); i++)
{ if cover in a[i]:
        Result = result + 1;
        if result > max:
            max = result;
   else:
        result = 0
        cover = I[i].endpoint;
} return max ;
```

<u>correctness</u>: I am sorting the intervals and taking endpoints to detect set of points that stabs all the Interval. Each time it finds a stab, it adds 1 to resource count. The stab points maintain the maximum number of intervals that needs resourses that overlaps. Endpoint makes sure that overlapping isn't overcalculated, and minimum number of resources are used. So the maximum intervals of each stab points is the minimum number of resources selected

First sort all the activities according to its end point. Run a while loop through all the activities until none left. Inside the while loop, run a for loop where it checks if any activities that doesn't overlap. If not overlaps, it removes the activities from the activities array and then increase the resource count by 1.

Amortized cost

i	1	2	3	⋯	101	102	⋯	201	202	⋯
size	1	101	101		101	201		201	301	
cost	1	1+1	1+0		1+0	1+101		1+0	1+201	

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=1}^{n/100} (100(j-1)+1) = n + 100\sum_{j=1}^{n/100} j - \sum_{j=1}^{n/100} 99 < n^2$$

$$A.P \Rightarrow \sum_{i=1}^{k} i = 1+2+3+4+\cdots+k = \frac{k(k+1)}{2}$$

$$H.N \Rightarrow \sum_{i=1}^{k} = \frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k} = \log k = \theta(\log k)$$

$$G.P \Rightarrow \sum_{i=0}^{k} x^i = x^0+x^1+x^2+\cdots+x^k = \frac{x^{k+1}-1}{x-1}$$

$$\infty\, G.P \Rightarrow \text{when } |x|<1 . \text{ eg } x=\tfrac{1}{2} ; \sum_{i=0}^{k} = \frac{1}{1-x}$$

let $\hat{c}_i = n$ for all i
consider the bank after k operations for any $k \in \{1 \ldots n\}$
Amount deposited $= k.n$
Amount withdrawn $< k + \sum_{j-1}^{k/100} k - 99 < k^2$
$k.n > k^2$ so bank is ≥ 0

#

i	1	2	3	4	5	6	7	8	⋯
size	1	4	4	4	16	16	16	16	
cost	1	2	1	1	5	1	1	1	

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=1}^{\log_4 n} 4^j = n + \frac{4^{\log_4 n+1}-1}{3} = \frac{7n-1}{3} = \frac{7}{3}n - \frac{1}{3}$$

→ single operation, $\hat{c}_i = \frac{7}{3} - \frac{1}{3n}$
we just can drop the $\frac{1}{3n}$ as it is too small for large number of n's. And take the ceiling of 7/3 which is 3. So amortized cost for single operation, $\hat{c}_i = 3$

#

i	2^0 1	2^1 2	3	2^2 4	5	⋯	2^3 8	9	⋯	2^4 16
cost	5	5*2	1	5*4	1	⋯	5x8	1	⋯	5x16

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=1}^{\log_2 n} 5 \cdot 2^i = n + 5 \times \frac{2(2^{\log_2 n+1}-1)}{1}$$
$$= n + 10(2n-1)$$
$$= 21n - 10 \quad \therefore O(n)$$

∴ The cost of single operation $= (21n-10)/n = 21 - \frac{10}{n}$
The $\frac{10}{n}$ is negligible for big n, so we remove it.
$\therefore \hat{c}_i = 21$

A directed graph is weakly connected if for every pair of vertices u; v we have that either there is a path from u to v or there is a path from v to u. Give an algorithm to determine if a given directed graph G is weakly connected. Make your algorithm as e

Initialize an nxn array A to all be false
→ For each vertex u, run BFS, and for each vertex V it finds, set A[u,v] to be true.
→ After running for all vertices, if there is a u,v such that A[u,v] and A[v,u] are false, then the graph is not weakly connected. otherwise it is. Running time : $\theta(n^2)$

Suppose G = (V, E) is a connected, undirected graph. That is, for any two vertices u, v € V, there is a path that connects u and v. An edge e € E is called a bridge if its removal disconnects the graph. Give an algorithm that determines if G contains an edge. Make your algorithm as efficient as possible. What is the running time of your algorithm?

→ I will run a loop over all the edges. Remove the edge. Then run BFS/DFS from both the vertices connected to the edge each time. If in any pair of vertices (x, y) we get no way to reach from x to y, we have a bridge. If we don't get so, we don't have any bridge.
Complexity : We are running traversal for every edge. So the running time would be $m(m+n) = m^2+mn = O(m^2)$ [As $m>n$ in most cases]

Suppose we have an undirected, unweighted graph G. For any two vertices u, u of G, let 6(u, r) denote the Jength of the shortest path between w and v where the length of the path is determined by the number of edges in the path. The diameter of a graph is defined to be mAX, 6(u, v). In other words, if we consider all pairs of vertices, it's the pair that maximizes their shortest path distance. Give an algorithm to compute the diameter of G. The algorithin can run in O(nm) time.

We can use BFS algorithm to determine the shortest path distance, since edges are unweighted. Run BFS for each node or vertex, remembering the longest shortest path we encounter.
→ This runs in O(nm) time

Suppose we are given an undirected graph G = (V, E) that represents a social media graph (vertices represent accounts and edges represent that the accounts are "friends"). For any vertex v € V, we can consider how many degrees of separation v is from some other vertex. If a vertex u is friends with v, then v and u have one degree of separation. If a vertex w is not friends with v but is friends with some friend of v, then u and w have two degrees of separation. In this problem, the goal is that given any vertex v, we want to find a vertex in the graph that maximizes the degrees of separation from v. Give an algorithm that finds such a vertex and runs in O(n + m) time.

We can run BFS and maintain a degree counter. We will run the BFS starting from v and mark every vertex that has a edge from v as the one degree friend
As we move to the adjacent vertex, we will update the degree counter for his adjacent vertex by adding 1 with the current vertexes degree of seperation.
Save the largest degree of seperation vertex in a variable and return it
The runtime for this algorithm is O(V+E) or O(m+n)

For this problem set I will use the Kruskal algorithm. It will automatically sort the edges in non-decreasing order, so when we find the minimum spanning cost, we just have to save the last cost to go from u to v and then use free edge coupon in it. We can achieve it by subtracting the last (maximum edge) and reducing it from the total cost.

In class we showed how to compute a minimum spanning tree and a shortest path in a graph. Suppose we want to do the opposite for both of them. That is, suppose we want to compute a Maximum Spanning Tree (spanning tree that maximizes sum of edge weights) and a Longest Path (path in the graph that maximizes the sum of edge weights where we visit each vertex once). Can we easily modify the algorithms that we have seen in class to handle these cases? If so how can this be done? If not then what is the di culty?

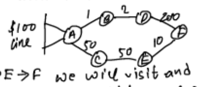
Max spanning tree can be computed by multipulying all edges weight by -1 and then runnig any MST algorithm on the new edge weight
Longest path unfortunately cannot be computed easily because the key property break down :
A subpath of a longest path may not be a longest path

Given a graph G = (V, E) with weights on the edges (positive and negative), let 5(u, v) denote the length of a shortest path connecting u and v. The diameter of G is defined to be max,,veV §(u, v) (note that the diameter is a number). Give a polynomial-time algorithm to compute the diameter of G. Your algorithm can use shortest path algorithms that we discussed in class as a subroutine of your algorithm. Make your algorithm as efficient as possible. What is the running

As there are some negative weight paths, we will use bellman-ford algorithm. I will run bellman ford algorithm for n times, for all the vertices. Then, we will get all the shortest possible paths for every pair of vertices. I will just pick the maximum one and that is the diameter of the graph G.
complexity : $n(mn) : mn^2$
If no negative path, use dijkstra Runtime $= n^2 + mn = O(mn)$

Suppose we have a directed graph G = (V, E) where for each edge (u, v) € E we have a positive weight on the edge w(u, v) that denotes how much money it costs to travel from u to v where the units are dollars (eg., if w(u, v) = 20 then this means it costs 820 to travel from u to v). We are interested in traveling from a vertex s to a vertex t for as cheaply as possible. This problem can be solved with Dijkstra's Algorithm as all of the edge weights are positive.
Now suppose we have a setting where there is a promotion where if you have already spent $100 on the path, then you can take one free trip (essentially you can make a single edge's weight be worth 0 on this path anytime after $100 has been spent). If you have a given path p from s to t, then it is fairly easy to figure out the best way to use the free edge: you would use it on the most expensive edge after $100 has been exceeded. It is more challenging to compute the path that leads to the minimum total cost in this setting.
A reasonable idea to design an algorithm for this problem would be to start with something similar to Dijkstra's Algorithm and consider what modifications need to be made to handle the new setting. Why might Dijkstra's algorithm fail on this problem without modification, and what modifications need to be made to be able to compute this style of shortest paths?

Let us consider a node that is in the path from S to t. Let's consider t' be the node that is $100 cost away from S. We will run dijkstra again from all the t' and figure out the shortest path from t' to t. We will keep max variable to record the max weighted edge and just subtract that from the total cost to get the minimum cost.
This will avoid heavy edges.



→ gives path that includes A→C→E→F we will visit and make one 50 go away, but the shortest path would been A→B→D→F after deleting that 200 edge. This is were dijkstra fails.
→ So possible solution would be to run dijkstra from t' making one edge zero in each iteration and figure out the shortest path that way. We can optimize it by making all weights 1 and then run BFS. This will figure out all the path leading from s to t. Then find t' using dijkstra, then from t' to t considering every path that leads to t.

Runtimes :

→ BFS and DFS : n+m

→ MST : Prims : Maintains a key for each vertex (initiall 0)
- Arbitrarily pick a vertex and changes its key to zero.
- Maintains one tree
- maintains binary heap datastructure
- runs in O(mlog n)

Kruskal's algorithm : Repeatedly pick the edge with the smallest weight as long as it does not form a cycle.
- Maintains a forest
- runs in O(mlog m)

→ Shortest path :
- To compute the actual path (instead just the lenght). We can use MST, BFS, DFS trees (just remember the predecessor for each vertex during computaion).
- Dijkstra : Can be used when there is no negative edge.
  - It is a greedy algorithm
  - Runs in O(mlog n)
  - If there is unweighted grap, we can set the weights to b 1.
  - Another way is to modify BFS search starting from s. Recall BFS traverses the graph in layers where each layer will be the same distance from S.

Single-source shortest paths:
{ Nonnegative edge weights. Dijkstra: O(mlog n)
{ Arbitrary edge weights. Bellman-Ford: O(nm)
{ DAG: Bellman-Ford single pass: O(n + m)

All-pairs shortest paths:
{ Nonnegative edge weights. Dijkstra n times: O(nmlog n)
{ Arbitrary edge weights. Bellman-Ford n times: O(n2m)

Such an ordering is called a topological sort.
One way of obtaining a topological sort is to perform a DFS and then sort the nodes in decreasing order according to nishing time.