1. The greedy algorithm we described for the class scheduling problem is not the only greedy strategy we could have tried. For each of the following alternative greedy strategies, either prove that the resulting algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control).

(a) Choose the course x that ends last, discard classes that conflict with x, and recurse.

This strategy doesn't work all the time. Assume the case where the below are the activity schedules

Activity	*A1*	*A2*	*A3*	*A4*	*A5*
Start	*1*	*2*	*5*	*3*	*10*
Finish	*4*	*6*	*8*	*23*	*16*

In this case, the algorithm chooses A4, instead of A1, A3, A5 which is the optimal schedule

(b) Choose the course x that starts first, discard all classes that conflict with x, and recurse.

This strategy doesn't work all the time. Assume the case where the below are the activity schedules

Activity	*A1*	*A2*	*A3*	*A4*	*A5*
Start	*5*	*4*	*3*	*11*	*15*
Finish	*10*	*6*	*16*	*14*	*22*

Here the algorithm chooses A3, instead A1, A4, A5 or A2, A4, A5 is the optimal schedule

(c) Choose the course x that starts last, discard all classes that conflict with x, and recurse.

This strategy works. There is an optimal schedule that includes the course 'x' that starts last.

Assume activities {A1, A2…An} with start times {S1, S2…. Sn} and finish times {F1, F2…. Fn}

and Ax is the course that starts last.

- *Let OPT be an optimal solution*
- *If OPT contains Ax we're done*
- *If not, let Ay be the last course in it, and Sy < Sx, as Ax starts the last.*
- *For every course i, if Fi < Sy< Sx, then (OPT - {Ay}) U {Ax} is still an optimal solution containing Ax.*

Clearly, an optimal solution for the subset of activity compatible with Ax plus Ax is an optimal solution.

(d) Choose the course x with shortest duration, discard all classes that conflict with x, and recurse.

This strategy doesn't work all the time. Assume the case where the below are the activity schedules

Activity	*A1*	*A2*	*A3*
Start	*1*	*8*	*5*
Finish	*7*	*16*	*9*

Here the algorithm chooses A3, instead A1, A2 is the optimal schedule

(e) If no classes conflict, choose them all. Otherwise, discard the course with longest duration and recurse.

This strategy doesn't work all the time. Using the above example

Activity	*A1*	*A2*	*A3*
Start	*1*	*8*	*5*
Finish	*7*	*16*	*9*

Here the algorithm discards A1 and A2 and chooses A3, instead A1, A2 is the optimal schedule.

(f) If no classes conflict, choose them all. Otherwise, discard a course that conflicts with the most other courses and recurse.

This strategy doesn't work all the time. Assume the case where the below are the activity schedules

Activity	*A1*	*A2*	*A3*	*A4*	*A5*	*A6*	*A7*	*A8*
Start	*1*	*2*	*5*	*3*	*9*	*10*	*13*	*11*
Finish	*4*	*6*	*8*	*7*	*12*	*14*	*16*	*15*

Here the algorithm discards either A3 or A5 and chooses A1, A3, A6 or another set of three courses but four courses A1, A3, A5, A7 is the optimal solution.

(g) If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that ends last, discard all classes that conflict with y, and recurse.

This strategy works. If a course Ax contains another course Az completely then there is an optimal schedule

- *Let OPT be an optimal solution containing Ax*
- *Then (OPT - {Ax}) U {Az} is still an optimal solution of the same size*
- *If no course contains any other course, then the class that ends last which is also the one that starts last, Ay, following problem (c), this is an optimal solution*

2. An independent set in a graph G is a set of vertices I in G such that no two vertices in I are adjacent (neighbors). The maximum independent set problem is, given a graph G, to compute an independent set in G of maximum size (maximum number of vertices). Pinocchio claims that he has a greedy algorithm that solves the maximum independent set problem. Pinocchio's algorithm works as follows. The algorithm initializes the set I to the empty set, and repeats the following steps: Pick a vertex in the graph with the minimum degree, add it to the set I, and remove it and all the vertices adjacent to it from the graph. The algorithm stops when the graph is empty. Does Pinocchio's greedy algorithm always produces a maximum independent set? Prove your answer (if it does, give a proof; if it does not, give a counter example, that is, a graph on which Pinocchio's algorithm does not produce a maximum independent set).

Greedy algorithm

```
I ← ∅
While G is not empty do
Let v be a vertex of minimum degree in G
I ← I ∪{v}
Remove v and its adjacent vertices from G
end while
Output I
```

Here in this algorithm, the idea of choosing a vertex with least degree comes from a simple intuition that if I choose a vertex with high degree, elimination at each step decreases a lot the number of vertices that could be in the independent set. So, choosing a least degree vertex at each step affects the size of the maximum independent set.

This Pinocchio algorithm works fine in most cases but does not always find an independent set of maximum size.

Like discussed above, the least degree vertex at each step may not be as simple to choose, if there are multiple vertices with same least degree. In such cases we have to choose the vertex with greater total degree, so the induces sub-graph will have less edges. Hence, we choose the vertex with greatest second degree. If the same degree resides at that level too, we traverse one more level. At every step there can be several nodes with minimum degree, and the choice can dramatically change the output.

3. Professor Gekko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana. The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations. The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient algorithm by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

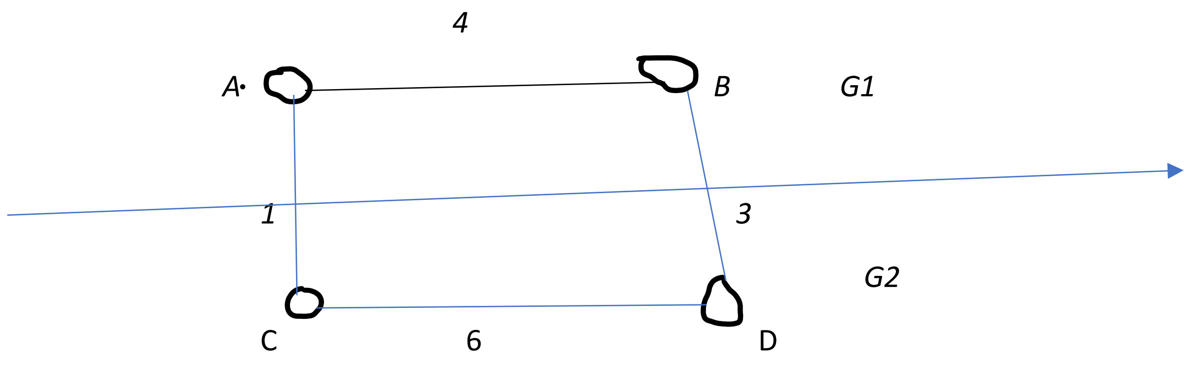
   We can optimize the solution to this problem using a greedy algorithm. At each water stop, he has to check whether he can make it to the next water stop without stopping at this one, he can skip it or not based on this. He has to reach the furthest point from the starting point which is less than or equal to 'm' miles before running out of water. This problem exhibits optimal substructure. If there are 'n' possible water stops, consider an optimal solution with 'p' stops made and first stop at 'o' location, then the optimal solution would be finding the solution to the subproblem with the remaining 'n-o' water stops.

   Let S be the optimal solution which stops made at s1,s2,s3…..sk. Let t1 be the first stop i.e furthest point we can reach from starting point. Then we can replace s1 by t2 to create a modified optimal solution V, since s2-s1 < s2-t1. As the new solution V has the same number of stops as S, t1 can be added in the optimal solution. So, by the greedy-choice property this strategy works.

4. Consider the following proposed divide-and-conquer algorithm for computing a minimum spanning tree. Given an undirected weighted graph G = (V, E), partition the set V into two subsets V1 and V2 such that the number of vertices in V1 and V2 differ by at most 1. Let E1 be the set of edges that are incident only on vertices in V1, and let E2 be the set of edges that are incident only on vertices in V2. Recursively call the algorithm on each of the two

subgraphs G1 = (V1, E1) and G2 = (V2, E2), and add the edge of minimum weight between the vertices of V1 and the vertices of V2 (i.e., an edge of minimum weight having one endpoint in V1 and the other in V2) to unite the resulting solutions from the two recursive calls. Does the proposed algorithm correctly compute a minimum spanning tree? Either prove the correctness of the algorithm or give a counter example (i.e., a graph) on which it fails.

We argue that the algorithm fails. Consider a graph G with four vertices A,B,C,D and is partitioned into subsets V1 and V2 such that V1={A,B} and V2={C,D}, E1={A,B}, E2={C,D}



While recursing through subgraphs G1 and G2, the min. spanning tree of G1 has weight 4 and MST of G2 is 6, the min. weight cutting the graph is 1. The spanning tree formed by the proposed algorithm is B-A-C-D has weight 11. But the MST of G, C-A-B-D has weight of 8 which is less. Hence the algorithm fails to get an MST.