Prob. 1	Prob. 2	Prob. 3	Prob. 4	Prob. 5

Problem 1.

We are going to show that the greedy algorithm is not optimal and as a ratio of 2 with the optimal solution.

The Greedy algorithm isn't optimal. The Figure 1 shows an input on which an optimal solution uses two unit length intervals, but our greedy algorithm uses three.

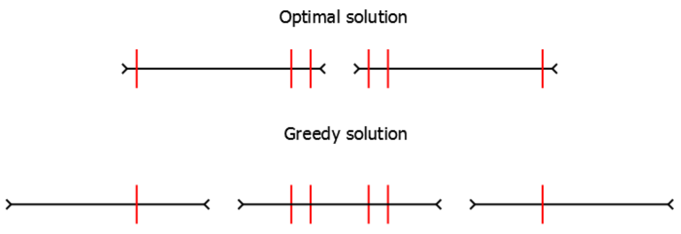

Figure 1: Greedy is not optimal

The Figure 2 shows a situation where our greedy algorithm output is as close as we want to 2 times the optimal solution. The idea is to create a lot of singletons that will not be recovered at first ad will result in a lot of overlap at the end. A naive algorithm that recovers all points without overlapping is nearly twice better in this situation.

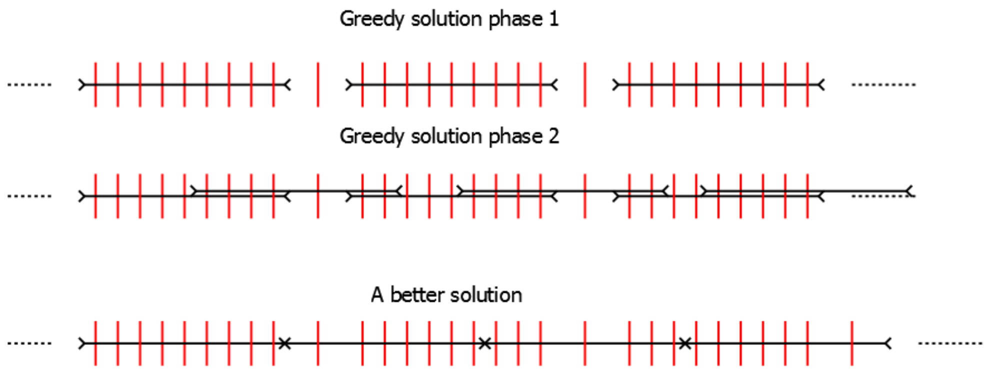

Figure 2: Greedy can be 2 times the optimal

The only thing left is to show that the greedy solution cannot be more than two times worst than the optimal.

We first realized that an optimal solution can be found using the simple rule : At all step we choose the interval that start with the leftmost not covered point. The optimality of this algorithm can easily be shown by induction considering an other optimal solution and showing that they use the same number of intervals.

We also consider a slightly different algorithm than the greedy. It chooses the same intervals except that it always puts a not-yet-covered point at the left extreme of the interval. This other version of the greedy algorithm builds the exact same number of interval.

We now want to show that our greedy algorithm does not use more than two times the number of intervals used above. We first show that any point cannot be covered by more than 2 intervals. This is illustrated by Figure 3. The interval 1 cannot be chosen after 2 or 3 because we follow the above rule, neither can 2 after 3. Furthermore 3 will always be chosen before 2 because 3 contains more uncovered points.

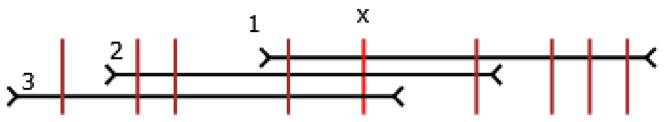

Figure 3: Greedy is not optimal

Problem 2.

The greedy algorithm described is not perfect because it will perform very badly on numbers of the form $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. For such numbers, the algorithm will make $k-1$ multiplication by 2 and then will make incrementations. But there will be $(2^k-1) - 2^{k-1} = 2^{k-1} - 1$ incrementations!

With the following greedy algorithm, that we will prove to be optimal, we will only do $k-1$ multiplcations by two and k incrementations for the same number.

Our algorithm that return for the number n the optimal sequence of Doubling/Incrementation operations in a list to get n (a "D" for a doubling operation, "I" for an increment operation) is the following:

```
copy n in m.
initialize the list l to empty.

While m is non zero do
    if m % 2 is zero then
        put a "I" at the beginning of l.
    end if
    put m/2 in m.
    if m is non zero then
        put a "D" at the beginning of l.
    end if
end while
return the list l.
```

When reading the list from the beginning to the right, we have the optimal sequence of Doubling/Incrementation operations to build the number n from 0. We can prove the correctness of our algorithm by induction because the set of positive integers can be defined by induction with: 0 is a positive integer, if m is a positive integer, $2m$ and $2m+1$ are too. Thus, if we prove that our algorithm is correct for 0 (it returns the empty list, which is correct) and for numbers written as $2m+1$ and $2m$ it will be correct:

- If we have the correct list l' of operations to build the number m with Doubling/incrementation operations, the returned list for $2m$ is l' with a "D" written in the last position (first step of the loop), and thus is correct.
- If we have the correct list l' of operations to build the number m with Doubling/incrementation operations, the returned list for $2m+1$ is l' with a "D","I" written in the last positions (first step of the loop), and thus is correct.

If we have a number q of 1s in the binary representation of the number n, any correct algorithm needs at least q incrementations (one incrementation adds at most one 1 in the binary representation). If our number n has it most significant bit in binary repsentation at the position k (starting from $k=0$), every correct algorithm needs at least k doubling operations. And our algorithm exactyl do q incrementations and k doubling, thus it is optimal.

Problem 3.

If k is the number of edges to remove from a graph to turn it into a tree, it means that the graph includes exactly k cycles: if there was more than k cycles, we would need to remove more edges to get a tree, if there was less than k cycles we would need to remove less edges.

Thus to get a minimum spanning tree, our algorithm needs to remove the k more expensive edges (no more or we will don't have a tree, no less or we will still have a graph) from the cycles. Our algorithm makes k rounds during which it will detect a cycle, and remove the most expensive edge from it.

Thus the important part is to detect one cycle in linear time, and remove the most expensive edge from it. By doing a Depth First Search from one of the edge in the graph, we can detect the cycle by: check if the other vertex (not the one from which we are coming from) is already marked, if yes, mark both vertices as in the cycle, continue recursion by marking the vertices as visited and following every edge. This DFS runs in linear time with respect to the number of edges in the graph. Then, after finding an edge in the cycle, we start another DFS to find the most expensive edge in the cycle not yet marked for deletion, which is still a linear time operation.

We have thus an algorithm to find the minimum spanning tree of a graph for which we know there are k cycles, which is running in $\Theta(k|E|)$ where $|E|$ is the number of edges in the graph.

Problem 4.

We assume we are given a maximum matching for connected bipartite graph of $2n+1$ vertices. Let's consider the component with the most vertices. In this component, there exists a vertex that is not part of the matching, call it v. Pick a edge from v : (u,v). We know that u is part of the matching (if not the matching is not maximal because we could add (u,v)). We replace in the matching the edge from u by (u,v) : we have created another maximum matching.

Therefore this graph does not have a **unique** maximum matching.

We are going to prove by induction that the maximum number of edges, a connected bipartite graph on $2n$ vertices, with a unique maximum matching, can have is : $\sum_{k=1}^{n} k$.

It's trivial for $n=1$.

Considering it is true until n, let's prove it for $n+1$. Let's consider the graph of $2n$ vertices with a unique and given matching. We add a vertex in each component. We can add $n+1$ edges for one of this two vertices to all the vertices in the other component. If we add more than $n+1$ edges we won't have a unique matching anymore.

Thus the maximum number of edges is $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$.

Problem 5.

Let's use here the maxflow-mincut theorem to be able to reasonne on a minimal cut instead of a flow. Let's consider we have a minimal cut for our problem, this cut is of value F. We denote for now on our cut by $\mathcal{C}$ and its cardinality (i.e. the number of edges in the cut) by $|\mathcal{C}|$.

If the capacity of every edge is increased by exactly 1 unit, the new mincut $\mathcal{C}'$, which is not necessarily the same set of edges, will have a new value $F+\Delta_1$. If our new mincut contained exactly the same set of edges as before, we would have $\Delta_1 = |\mathcal{C}|$. Therefore, $\Delta_1 \leqslant |\mathcal{C}|$.

If the capacity of every edge is decreased by exactly 1 unit, the new mincut $\mathcal{C}''$, which is not necessarily the same set of edges, will have a new value $F-\Delta_2$. If our new mincut contained exactly the same set of edges as before, we would have $\Delta_2 = |\mathcal{C}|$. Therefore, $\Delta_2 \geqslant |\mathcal{C}|$.

Thus, $\Delta_1 \leqslant |\mathcal{C}| \leqslant \Delta_2$.