Set 1

Platon
Karageorgis
P3180088

(1) a) Show how we can use Euclid(a,b) algorithm to compute the least common multiple of 2 positive integers $a, b > 0$.
b) In a RSA system, Oscar intercepts a ciphertext which is $C=10$ and was sent to Bob with a public key $e=11$ and $n=91$. Find the plaintext.

---

a) The Euclidean algorithm states that, given a, b we can compute the greatest common divisor g of a,b (the exact algorithm shouldn't concern us in this problem).
There is a formula that connects gcd with the least common multiple which says:

$$LCM(a,b) = \frac{a \cdot b}{gcd(a,b)} \quad (1)$$

So, after running Euclid's algorithm we obtain the only unknown variable needed to compute the least common multiple, which is the gcd. Assuming that the calculations are correct, we substitute a, b and gcd(a,b) in the formula and we obtain the value of the lcm.

b) From theory we know that:

$$M = C^d \bmod n \quad (1)$$

$$d \cdot e = 1 \bmod \phi(n) \quad (2)$$

The first equation if we substitute the known values is:

$$M = 10^d \bmod 91 \quad (3)$$

So if we can find d we will be able to compute M as well. We focus on the (2) formula. After the substitutions we have:

$$d \cdot 11 = 1 \bmod \Phi(n) \quad (4)$$

We need to find $\Phi(n) = (p-1)(q-1)$ where $p \cdot q = n$. In an average case, it would not be possible to calculate effectively p, q without knowing the secret key but in this case n is so small that we can easily spot it.

$n = p \cdot q$, where p, q are primes
$91 = p \cdot q$
Pick the first 5 primes: $\cancel{2}, \cancel{3}, \cancel{5}, 7, \cancel{11}$
91 is divided only by 7 and the remainder is 13 which is also a prime number. Therefore, $\Phi(n) = 6 \cdot 12 = 72$ (5)

$(4) \overset{(5)}{\Rightarrow} \quad d \cdot 11 = 1 \bmod 72$

We will now apply the Extended Euclidean algorithm to find d.

$72 = 6 \cdot 11 + 5$
$11 = 2 \cdot 5 + 1 \qquad (3) \Rightarrow \boxed{M = 10^{13} \bmod 91}$
$11 - 2 \cdot 5 = 1$
$11 - 2 \cdot (72 - 6 \cdot 11) = 1$
$11 - 2 \cdot 72 + 12 \cdot 11 = 1$
$-2 \cdot 72 + 13 \cdot 11 = 1$

So d is 13 and lastly we will use repeated squaring to find M

$10^1 \bmod 91 = 10$
$10^2 \bmod 91 = 9$
$10^3 \bmod 91 = 10^2 \cdot 10 \bmod 91 = 10 \cdot 9 \bmod 91 = 90 \bmod 91 = 90$
$10^4 \bmod 91 = 10^2 \cdot 10^2 \bmod 91 = 9 \cdot 9 \bmod 91 = 81 \bmod 91 = 81$
$10^5 \bmod 91 = 10^2 \cdot 10^3 \bmod 91 = 9 \cdot 90 \bmod 91 = 82$
$10^6 \bmod 91 = 10^3 \cdot 10^3 \bmod 91 = 90 \cdot 90 \bmod 91 = 1$
$10^7 \bmod 91 = 10^3 \cdot 10^4 \bmod 91 = 90 \cdot 81 \bmod 91 = 10$

$\vdots$ (We don't need the rest)

$10^{13} \bmod 91 = 10^6 \cdot 10^7 \bmod 91 = 1 \cdot 10 \bmod 91 = 10$

Finally, we can say that $\boxed{M = 10}$

(2) A frog is standing on a water lily. The water lilies are on a straight line, from 1 to n and the frog is standing on number 1. The frog will start jumping from one water lily to the other to eat the food located at water lily n. The maximum jump the frog can make is different for each starting water lily and is supposed to be available in an available jump[1...n] matrix. We assume that jump[i] ≥ 1 for each $i \in \{1, 2, \ldots, n\}$. The goal is to find the least number of jumps the frog has to make in order to reach the nth water lily

a) Prove with a counter-example that the greedy solution choosing always the maximum leap will not be optimal in every case.

b) Design a dynamic programming algorithm for the problem and calculate its complexity.

---

a) 1 2 3 4 5 6
× × × × × ×

jump[i] = [3, 2, 3, 1, 1]
(assume it starts from 1)

In a simple example, if the frog implements the greedy method for the values above it will have to make 1 jump to reach water lily number 4, one more to reach 5 and a last one for the $n_{th} = 6_{th}$ water lily. So in total 3 jumps. We can see that the optimal solution is 2, since there is the choice to jump from the first water lily to the 3rd water lily and from that one directly to water lily number 6. Therefore, greedy indeed doesn't always return the optimal solution.

b) Since we want a dynamic programming algorithm a first step would be to identify the variables that are required in each sub-problem. The two variables are, the position, and the distance that the frog can jump. But, we have the table jump[i] available and the only thing we need to access it is the position. So, we conclude that the only variable that is needed, is the position. Subsequently, we analyze the cases of the problem. The first case is that the frog hasn't reached the final waterlily and the second one is that the n_{th} waterlily has been reached. Lastly, we create the recursive function while keeping in mind that the goal is to minimize the number of jumps.

$$OPT[i] = \begin{cases} 0, & i = i_{max} \\ \min\{frog[jump[i]+1]\}, & i < i_{max} \end{cases}$$

We don't need to consider the case where $i > i_{max}$ because we expect the algorithm to terminate when $i = i_{max}$ and i increments with step = 1. The logic behind this function is that we collect in each step all the possible jumps from the ith waterlily, and we are able to calculate a result as soon as we get the values in range (i+1, k) where k is the maximum jump from i which is the value stored at jump[i]. The complexity of the algorithm is $\underline{O(n^2)}$ because what we are actually doing is a nested loop where n is the size of the jump table.

④ Assume we have the following problem. Given a set (universe) U consisting n elements, each having a non-negative weight. We also have a set $S = \{S_1, S_2, \dots, S_l\}$ where each S_i is a subset of U and a positive $k \leq l$, and we are asked to pick k sets from the collection S so that the total weight of the covered elements of U is maximised. Given the obvious greedy algorithm: "As long as we don't have k sets from S, add to our solution the best possible set" (where the best set is the one with the elements with the biggest weight).
Prove that this algorithm has an approximation factor:

$$\cancel{1} - \left(1 - \frac{1}{k}\right)^k > \cancel{1} - \frac{1}{e} \Rightarrow \boxed{\left(1 - \frac{1}{k}\right)^k < \frac{1}{e}}$$

Let's assume a set T consisting the elements that are uncovered in our solution and covered in the optimal solution. We have opt as the optimal solution, which obviously has collected the maximum weight and we define l_i as the difference of our solution and opt. It can be deduced that $T \leq l_i$, since in the reversed case where $l_i < T$, the optimal solution wouldn't be optimal. Since our solution covers the elements with k sets, then by following the same pattern there must be a set that covers:

$$\frac{T}{k} \leq \frac{l_i}{k}$$ new elements. Therefore for each i we must have:

$$l_i - l_{i+1} \geq \frac{l_i}{k}$$ which leads us to the following recurrence:

$$l_k \leq l_{k-1} \cdot \left(1 - \frac{1}{k-1}\right)^1 \leq l_{k-2}\left(1 - \frac{1}{k-2}\right)^2 \dots \leq l_0 \cdot \left(1 - \frac{1}{k0}\right)^k$$

which is equal to: $l_k \leq l_0 \cdot \left(1 - \frac{1}{k0}\right)^k = OPT \cdot \left(1 - \frac{1}{k}\right)^k$

Finally, it can be seen that: $\boxed{OPT\left(1 - \frac{1}{k}\right)^k < OPT\left(\frac{1}{e}\right)}$

(5) Assume the following algorithm for Vertex Cover in an undirected graph without weights, $G=(V,E)$:
Find a DFS tree T in G and return the set S consisting all the vertices that are not leaves of T.
Prove that S is a vertex cover and that $|S| \leq 2 \cdot OPT$, where OPT is the cost of the optimal solution of Vertex Cover.

We must show that S is a vertex cover. We will prove it by contradiction. Let's say it's not a vertex cover. Then, there must be an edge n that is connected to 2 vertices (u,v) and none of those belongs in S. A tree has 3 different types of edges.

(i) edge that connects the root with the internal node
(ii) edge that connects 2 internal nodes
(iii) edge that connects an internal node with a leaf

Since we got in S every internal node it is obvious that in every case the edges are covered. We conclude that the edge n does not exist and therefore S is a vertex cover.

Now we have to show that the solution approaches the optimal with an approximation factor 2.
The key idea is that we want to "connect" our problem to the matching problem.

Step 1

Let's say that we have a matching of size n. A vertex cover must also be at least of size n. If the cover had a size $n-1$ or less then there would be at least one edge on the matching that would be uncovered. This statement is relatively easy to prove, since none of the n edges of the matching have a common vertex. So, if the vertex cover had a size $n-1$ then 1 edge would be uncovered.
We can say that, $|V| \geq |M|$ (1)

Step 2

Moving on, let's assume that we have a maximal matching instead of a normal matching like in the previous step. This matching has V_0 vertices and V_1 minimum vertex cover. It is obvious that V_0 is a vertex cover since we would have at least one uncovered edge otherwise, and that is not possible, because we assumed that we have a maximal matching. So, $|V_0| = 2|M|$ where V_0 is a vertex cover and from (1) we have: $|V_0| \leq 2|V_1|^{(2)}$ since $|V_1| \geq |M|$.

So now what? All we have to do is to show that $|S|$ can be V_0. We start picking edges from S until there is a maximal matching. This can obviously be performed in polynomial time. Therefore, from (2) since V_1 is a minimum vertex cover we conclude that: $|S| \leq 2 \cdot OPT$

3) We have the following encoding problem:

$\boxed{\text{Input}}$ An $n \times m$ matrix $A = (a_{ij})$ with elements 0 or 1, a vector $y^T = [y_1, y_2, \dots, y_m]$ with elements 0 or 1 and an integer $k > 0$.

$\boxed{\text{Question}}$ Is there a vector $x^T = [x_1, x_2, \dots, x_n]$ with elements 0 or 1 such that (1) x has at most k 1's and (2) $1 \leq j \leq m$, $\sum_{i=1}^{n} x_i \cdot a_{i,j} \equiv y_j \pmod 2$?

You can think A as a decoding matrix of a linear code, y as a binary message and x as the encoding of the message based on the given code. We are looking for an encoding with a small number of 1's. Show that this problem is NP-Complete.

First we want to show that our problem belongs in NP.
Given an assignment for x we can use XOR-SAT which can be viewed as a system for linear equations mod 2, and decide if there exists a solution. This happens in polynomial time and so we conclude that our problem belongs in NP. (1)

Now we want to find a NP-Complete problem that can be reducted to ours. We are going to use 3-SAT for our proof. The reduction is a bit complicated so we will do it in steps and explain in each step.

1. For each element y_j in the vector y^T, create a Boolean variable y_j.
2. For each element x_i in the vector x^T, create a Boolean variable x_i.
3. For each row i in the matrix A and each element a_ij, create a Boolean variable a_ij.

Now that we have defined our 3 literals, we continue with the required constraints of our problem

4. For each column j in the matrix A, create a clause (y_j OR a1_j OR a2_j ... OR an_j). This clause enforces the condition that at least one element of x must be 1 (remember that $\sum_{i=1}^{n} \dots$ starts from 1).
5. For each column j in the matrix A and each element a_ij, create a clause (~y_j OR ~a_ij). This clause enforces the condition that if an element of x is 0, then the corresponding element of the matrix A must also be 0 in order to satisfy the given equation

The first demand of our problem should be fully fullfilled by our implementation, which is the equation $\sum_{i=1}^{n} x_i \cdot a_{ij} \equiv y_j \pmod 2$. We move on to the next constraint.

6. For each element x_i in the vector x, create a clause (x_i OR x_{i+1} OR ... OR x_k). This clause enforces the condition that we want up to k 1's. If there is a solution with at most K 1's then the clause will be True. Otherwise, it will be false. In general, if we want to minimize the number of 1's then we can ask repeatedly until the algorithm returns True for k and false for k-1. Obviously, we can't do the optimization version at once because then the problem would not be NP-Complete. The reduction is complete since we matched the inputs of the two problems. If we have a YES-instance for 3-SAT, then we can use the corresponding values for the literals in order to satisfy our constraints. If we have a YES-instance for our problem then we can extract the values of our variables and solve 3-SAT. Therefore, our problem is NP-Complete if we combine (1) with 3-SAT $\leq_p$ Encoding Problem. On a final note, the size of the resulting 3-SAT formula is $O(n \cdot m)$ which is clearly within the boundaries of polynomial time, so our reduction is valid