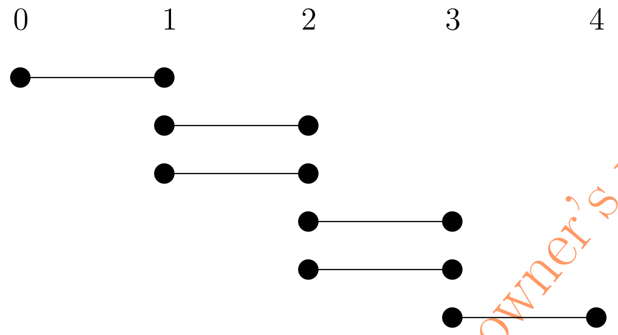
Computer Science C73
Scarborough Campus

September 14, 2022
University of Toronto

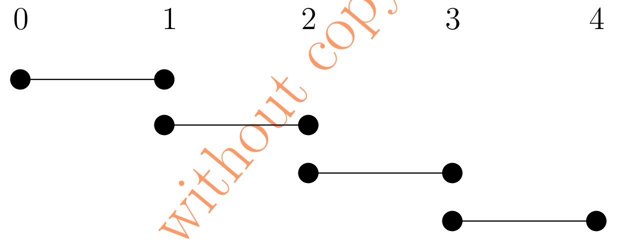
Solutions for Homework Assignment #1

Answer to Question 1.

a. The figure below shows two counterexamples.



Counterexample 1



Counterexample 2

In Counterexample 1 there is a unique number that is contained in the maximum number of intervals, namely 2. It intersects the two intervals $[1, 2]$ and the two intervals $[2, 3]$, so after these are removed we are left with two intervals $[0, 1]$ and $[3, 4]$. Because these intervals are disjoint, two more points (one in each of these two intervals) are needed to complete the cover. So (no matter how ties in the choice of c are resolved), the algorithm returns a cover consisting of three numbers. However, $\{1, 3\}$ is a smaller cover, so the algorithm returns a non-optimal cover in this case.

Counterexample 2 is simpler but does not have the desirable feature of causing the algorithm to return a non-optimal cover regardless of how the choice of c is resolved. There are three numbers that are contained in the maximum number of intervals, namely 1, 2, and 3. If the algorithm first choses 2, it will then remove the intervals $[1, 2]$ and $[2, 3]$, leaving two disjoint intervals $[0, 1]$ and $[3, 4]$, so it will return a suboptimal cover with three numbers (as there is a cover with only two numbers, namely $\{1, 3\}$). If the algorithm first choses 1 or 3, however, it will return the optimal cover $\{1, 3\}$. So, this is not as convincing a counterexample.

b. Our algorithm is as follows:

```
1 $C := \varnothing$
2 while $\mathcal{I} \neq \varnothing$ do
3     let $c$ be the minimum right endpoint of all intervals in $\mathcal{I}$
4     $C := C \cup \{c\}$
5     delete from $\mathcal{I}$ all intervals that contain $c$
6 return $C$
```

Correctness. First we observe that the while loop terminates; this is because in each iteration at least one interval is deleted from $\mathcal{I}$. Therefore the algorithm computes a set C of numbers.

Let $c_1 \leq c_2 \leq \ldots \leq c_k$ be the elements in C computed by the algorithm. Let $c_1^* \leq c_2^* \leq \ldots \leq c_m^*$ be the elements in some optimal cover C^* of $\mathcal{I}$. It is straightforward to show the following invariant: At the end of each iteration,

(1) C contains a cover of all the sequence of intervals deleted from the given sequence of intervals $\mathcal{I}$.

By (1) and the loop exit condition, the set C computed by the algorithm is a cover of $\mathcal{I}$. It remains to show that $k \leq m$. First we prove a "greedy-stays-ahead" type of invariant:

(2) For each i, $1 \leq i \leq m$, $c_i \geq c_i^*$.

We prove (2) by induction. The basis, $i = 1$, is obvious since otherwise C^* has no element contained in the interval in $\mathcal{I}$ that has minimum right endpoint and therefore it is not a (minimum) cover.

For the induction step, suppose that $c_i \geq c_i^*$ for some i, $1 \leq i < m$. We will prove that $c_{i+1} \geq c_{i+1}^*$. Suppose, for contradiction, that $c_{i+1} < c_{i+1}^*$. Consider iteration $i+1$, in which c_{i+1} is added to C. By the algorithm, c_{i+1} is the right endpoint r of some interval $I = [\ell, r]$ that is in $\mathcal{I}$ at the end of iteration i.

By (1), I does not contain any of $c_1, c_2, \ldots, c_i$ (otherwise it would have been removed from $\mathcal{I}$ by the end of iteration i). Therefore $c_i < \ell$. Since $\ell \leq r = c_{i+1} < c_{i+1}^*$ and (by induction hypothesis) $c_i^* \leq c_i$, we have

$$c_i^* \leq c_i < \ell \leq r = c_{i+1} < c_{i+1}^*.$$

But this means that the interval $I = [\ell, r]$ starts after c_i^* and ends before c_{i+1}^*, so it does not contain any element of C^*, contradicting that C^* is a (minimum) cover of $\mathcal{I}$. This completes the proof of invariant (2).

Now we use (2) to prove that C is a minimum cover. As argued earlier, C is a cover of $\mathcal{I}$. Suppose, for contradiction, that C is not a minimum cover. Since it is a cover, $m < k$. Consider the iteration $m+1$ in which c_{m+1} was added to C. By the algorithm, c_{m+1} is the right endpoint r of some interval $I = [\ell, r]$ that is in $\mathcal{I}$ at the end of iteration $m+1$. By (1), I does not contain any of $c_1, c_2, \ldots, c_{m+1}$ (otherwise it would have been removed from $\mathcal{I}$ by the end of iteration $m+1$). So, we have that $c_m^* \leq c_m < \ell \leq r = c_{m+1}$. Since c_m^* is the maximum element in C^*, I contains no element of C^*, contradicting that C^* is a (minimum) cover.

Answer to Question 2.

Algorithm: We consider the inverse process: Starting with $x = n$ go to $x = 1$ by applying a sequence of operations of the following two kinds: decrement x by 1 (i.e., $x := x - 1$) and halve x (i.e., $x := x/2$) provided that x is even. (Note that the double operation always results in an even number, so it makes sense that its inverse can be applied only if x is even.) Clearly, S_n is a sequence of operations for the process starting with 1 and ending with n if and only if the sequence R_n of the ***inverse*** operations in the ***reverse*** order is a sequence for the inverse process starting with n and ending with 1. Thus, a shortest sequence of operations for the inverse process corresponds to a shortest sequence of operations for the forward process, by executing the inverse of each operation in the reverse order.

We claim (and we will later prove) that the following greedy algorithm results in a shortest sequence of operations for the inverse process: If x is even, apply the halve operation; otherwise apply the decrement operation. Thus, a greedy algorithm to solve our problem is as follows. (Intuitively, we greedily apply the halving operation when that is possible — i.e., when x is even — thereby reducing x as much as possible; when we have no option, we apply the decrement operation.)

```
OptSeq($n$)
$x := n$
$S :=$ empty sequence
while $x \neq 1$ do
    if $x$ is even then
        $x := x/2$
        prepend D to $S$
    else
        $x := x - 1$
        prepend I to $S$
return $S$
```

Running time analysis: The running time of this algorithm is $O(\log n)$. Every at most two iterations we must apply a halving operation: If we apply a decrement operation, x is odd, and so in the next iteration x is even and we apply a halving operation. Thus, every at most two operations, x is reduced by at least half. So, the while loop is executed at most $2 \log_2 n$ times, and this dominates the running time of the algorithm.

Correctness: In what follows, instead of considering the sequence of operations that transform x from 1 to n, we consider the reverse sequence of the inverse operations, which transforms x from n to 1. Let $S^*(n)$ be the minimum number of operations required to transform n to 1, and $S(n)$ be the number of operations that our algorithm produces to transform n to 1.

We claim that for all positive integers n, $S(n) = S^*(n)$. Suppose, for contradiction, that for some positive integer n, $S(n) > S^*(n)$. By the well-ordering principle, there is a minimum such positive integer, say m. Obviously $m > 2$ (because for $m = 1$ there is only one sequence, the empty one; and for $m = 2$ there are two sequences, both of length 1: increment or double).

It must be that m is even, say $m = 2k$; otherwise only a decrement operation can be applied to m, resulting in $m - 1$, and by definition of m, our algorithm is optimal for $m - 1$. Therefore we have:

$$\begin{aligned}
S^*(m) &= S^*(2k) \\
&= 1 + S^*(2k-1) && \text{[since the optimal strategy for } 2k \text{ is different than in our algorithm,]} \\
& && \text{[and therefore it is to decrement, rather than halve]} \\
&= 2 + S^*(2k-2) && \text{[since } 2k-1 \text{ is odd]} \\
&= 3 + S^*(k-1) && \text{[since } 2k-2 < 2k = m \text{ and therefore our algorithm is optimal,]} \\
& && \text{and since } 2k-2 \text{ is even our algorithm applies the halve operation]} \\
&= 3 + S(k-1) && \text{[by definition of } m\text{, since } k-1 < m]
\end{aligned}$$

On the other hand, we have:

$$\begin{aligned}
S(m) &= S(2k) \\
&= 1 + S(k) && \text{[since our algorithm applies the halve operation to even numbers]} \\
&= 1 + S^*(k) && \text{[since } k < 2k = m \text{ and therefore our algorithm is optimal]} \\
&\leq 2 + S(k-1) && \text{[since one possible strategy is to apply a decrement operation to } k] \\
& && \text{[and then follow our algorithm]}
\end{aligned}$$

From the above two inequalities we get

$$S(m) \leq 2 + S(k-1) < 3 + S(k-1) = S^*(m) \qquad \Longrightarrow \qquad S(m) < S^*(m),$$

which contradicts the definition of $S^*(m)$.