CSE 6331 HOMEWORK 7

BRENDAN WHITAKER

1. *Let $A_n = \{ a_1, ..., a_n \}$ be a set of distinct coin types (e.g., $a_1 = 50$ cents, $a_2 = 25$ cents, $a_3 = 10$ cents, etc). Note that a_i bay be any positive integer and $a_1 > a_2 > ...a_n$. Each type is available in unlimited quantity. Given A_n and an integer $C > 0$, the coin changing problem is to make up the exact amount C using a minimum total number of coins.*

   (a) *Show that if $a_n \neq 1$, then there exists an A_n and C for which there is no solution to the changing problem.*
   Proof. Let $A_n = \{ 2 \}$ and let $C = 1$. We have $n = 1$ and $a_n = a_1 = 2 \neq 1$, and there is clearly no solution since no integer multiple of $a_n = 2$ will give us $C = 1$. $\square$

   (b) *Show that if $a_n = 1$ then there is always a solution.*
   Proof. Note since $C \in \mathbb{Z}$, and $a_n = 1$ we just take C coins of type a_n, and since we have an unlimited quantity available, we have a solution. $\square$

   (c) *When $a_n = 1$ a greedy method to the problem will make change by using coin types in the order $a_1, a_2, ..., a_n$. When coin type a_i is being considered, as many coins of this type as possible will be used. Show that this algorithm doesn't necessarily generate an optimal solution.*
   Proof. Let $C = 8$, and let $A_n = \{ 5, 4, 1 \}$. Our greedy algorithm will use one 5 cent coin, then three 1 cent coins. This is 4 total coins. But this is not optimal since we have a better solution using two 4 cent coins. $\square$

   (d) *Prove that if $A_n = \{ k^{n-1}, k^{n-2}, ..., k^0 \}$ for some $k > 1$, then the above greedy method always yields an optimal solution.* ***Hint:*** *Let $X = (x_{n-1}, ..., x_1, x_0)$ be the greedy solution and let $Y = (y_{n-1}, ..., y_1, y_0)$ be any optimal solution such that:*
   $$C = \sum_{i=0}^{n-1} x_i k^i = \sum_{i=0}^{n-1} y_i k^i.$$
   Show that $x_i = y_i$ for all $0 \leq i \leq n - 1$. Note: $k^m = 1 + \sum_{i=0}^{m-1} (k-1)k^i$.
   Proof. We prove by induction on n. Let $n = 1$. Then $x_0 = y_0 = C$ since $k^0 = 1$. Now fix $n = l \in \mathbb{N}$. Suppose $x_i = y_i$ for all $0 \leq i \leq l - 1$. We prove that for $A_{l+1} = \left\{ k^l, k^{l-1}, ..., k^0 \right\}$, $x_i = y_i$ for all $0 \leq i \leq l$. Suppose for contradiction that $x_l \neq y_l$. We can't have $x_l < y_l$ since the greedy algorithm uses as many coins as possible of type k^l. So we must have $x_l > y_l$. So set $x_l = y_l + j$. But since $k^i | k^l$ for all $i < l$, we know $\sum_{i=0}^{l-1} y_i \geq kj + \sum_{i=0}^{l-1} x_i$, since the option using the least amount of coins is by representing all jk^l cents (the amount by which

Date: SP18.

the sum of values of the first coin type differs between X and Y) as k^{l-1} cent coins. But then

$$\begin{aligned}\sum_{i=0}^{l} y_i &\geq x_l - j + kj + \sum_{i=0}^{l-1} x_i \\ &= (k-1)j + \sum_{i=0}^{l} x_l.\end{aligned} \tag{1}$$

And thus Y uses more coins than X, which is impossible since Y is optimal, so we have a contradiction. Thus we must have that $x_l = y_l$. Then by our induction hypothesis, we have a problem with $C' = C - x_l k^l$ and $l-1$ coins, for which we know $x_i = y_i$ for all $i \leq l-1$. Thus $x_i = y_i$ for all i with any $n \in \mathbb{N}$, and so the greedy method always yields an optimal solution. □

2. *Let $G = (V, E)$ be an undirected graph. A subset $U \subseteq V$ is called a* ***node cover*** *if each edge in E is incident upon at least one node in U. Finding a minimum node cover for a general graph is NP-hard, but if the graph is a tree, then a minimum node cover can be obtained by the greedy method. Design a greedy algorithm that always generates an optimal solution. (Explain your algorithm in plain English.)*

Algorithm 1: Node-Cover(v)

Data: The tree T, set U.
Result: A complete and optimal node cover U.

```
begin
    initialize C ⟵ Children(v);
    for c ∈ C do
        initialize C′ ⟵ Children(c);
        if C′ ≠ ∅ then
            Add c to U;
            Node-Cover(c);
        end
    end
    return;
end
```

We have a global variable T for the tree and a global set U initialized to the empty set, and we assume our program already has access to these. Our initial call is on v, the root of T. We set C to be the set of children of v. For each child, we set C' to be the children of c and if this set is nonempty, we add c to U, the node cover, and call Node-Cover(c). Thus it adds every vertex but the leaf nodes, which is exactly the optimal node cover for a tree.

3. *Consider the Activity problem discussed in class. Suppose now we want to maximize the* ***total sum*** *of selected intervals, $\sum_{i \in A}(f_i - s_i)$, where A is the set of selected intervals. Solve this problem using any method. Your algorithm must be $O(n^2)$.*

We assume the intervals are sorted by finish time, which takes $O(n \log n)$ time. We also assume all start and finish times are positive and real-valued. Define $F(i)$ to be the maximum total sum of selected intervals with finish time $\geq f_i$. We define:

$$F(i) = \max_{i<j\leq n+1} \begin{cases} (f_i - s_i) + F(j) & \text{if } s_j \geq f_i \\ F(i+1) \end{cases}. \tag{2}$$

Our boundary conditions are:

$$\begin{aligned} s_{n+1} &= \infty \\ F(n+1) &= 0. \end{aligned} \tag{3}$$

Our goal is to compute $F(1)$. We give an algorithm to compute our goal:

Algorithm 2: Activity-Compute-F

Data: Arrays of start times and end times $S[1..n+1], E[1..n]$.
Result: The array $F[i]$ computed for $1 \leq i \leq n$.

begin
  global array $F[1..n+1]$;
  initialize $F[n+1] \longleftarrow 0$;
  initialize $S[n+1] \longleftarrow \infty$;
  for $i \leftarrow n$ **to** 1 **do**
    $F[i] \longleftarrow \max_{j:S[j]\geq F[i]} \{ E[i] - S[i] + F[j], F[i+1] \}$;
  end
  return F;
end

And we give an algorithm to print out the set of selected intervals A:

Algorithm 3: Activity(i)

Data: The computed array $F[1..n+1]$, and the arrays of start times and end times $S[1..n], E[0..n]$, global set A.
Result: Computes the set A of selected intervals.

begin
  global set A;
  `/*` Assume $F[1..n+1]$ has already been computed. `*/`
  if $i < n+1$ **then**
    if $F[i] = E[i] - S[i] + F[i+1]$ **then**
      Add i **to** A;
      Activity($i+1$);
    else
      Activity($i+1$);
    end
  end
  return A;
end

The sort takes $n \log n$ time, the computation of F takes $O(n^2)$ time, since the max over j takes at most $O(n)$ time where $i = 1$, and the computation of A from F takes $O(n)$ time. So the whole algorithm takes $O(n^2)$ time.