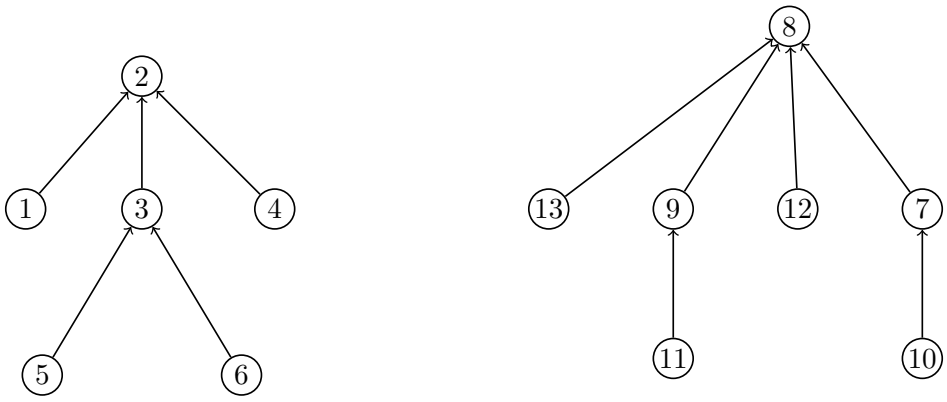
Homework 2

Due: September 20, in class.

Instruction. There are 9 problems in this homework, worth 100 points in total. For all algorithm design problems, you need to provide proofs on the correctness and running time of the algorithms unless otherwise instructed.

Problem 1 [12 pts]

(a) Recall that $union(A, B)$, links A to B (B being the parent) if A has less nodes than B; B links to A otherwise. Find a sequence of $union(.,.)$ operations to form the outcome in the figure below (assuming that you start from the scenario where every element is in a distinct set).



(b) We would like to union the set containing Element 5 with the set containing Element 11. Please use two $find(.)$ operations and one $union(.,.)$ operation to achieve this.

(c) Draw the resulting structure when

(a) $find(.)$ is not implemented with path compression.

(b) $find(.)$ is implemented with path compression.

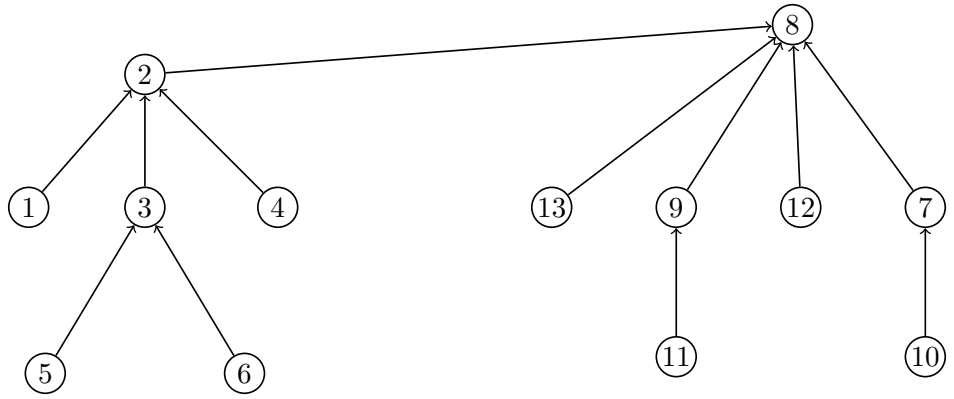
Solution:

(a) We can construct the left subtree first. One possible solution is $union(1, 2)$, $union(4, 2)$, $union(5, 3)$, $union(6, 3)$, $union(2, 3)$, $union(13, 8)$, $union(12, 8)$, $union(11, 9)$, $union(9, 8)$, $union(10, 7)$, $union(7, 8)$ in sequence.

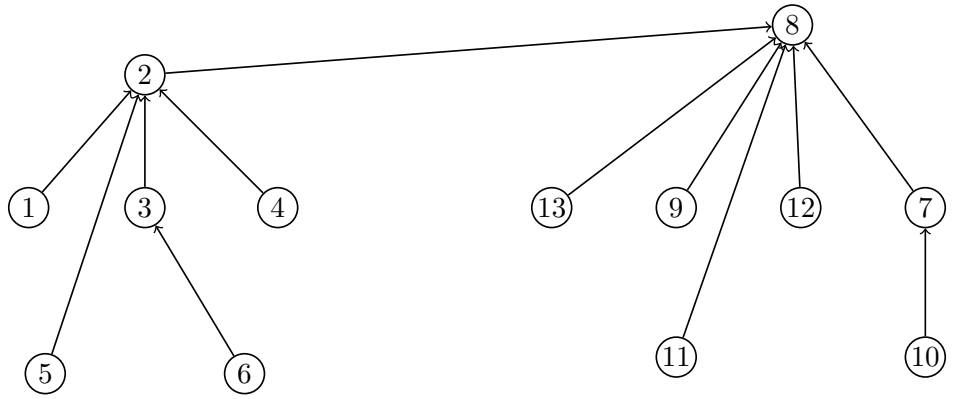
(b) Here is the procedure.

1: $p_1 \leftarrow find(5)$
2: $p_2 \leftarrow find(11)$
3: $union(p_1, p_2)$

(c) (a) Here is the resulting figure.



(b) Here is the resulting figure.



Problem 2 [12 pts]

There are n monkeys, numbered from $1, 2, 3, \cdots, n$. Monkeys may hold another monkey or a branch of a tall tree (the branch is denoted by number 0). This information is given to you in the form of m pairs (u_i, v_i) $(i = 1, 2, 3, \cdots, m)$, meaning that monkey u_i holds monkey v_i or the branch (if $v_i = 0$). Note that a monkey may hold more than one monkeys or the branch as it has 4 limbs and a tail.

Now you are given T pairs (u_t, v_t) $(t = 1, 2, 3, \cdots, T, T \leq m)$, each of which appeared in the m pairs above. At the t^{th} second, monkey u_t releases its grasp on v_t. And this might lead to the fall of several monkeys on the ground.

Design an algorithm that runs in time $O(m\alpha(n))$ to compute the time when each monkey i falls on the ground, or that it does not fall after all T seconds. You will receive partial credit for algorithms in time $O(nm)$.

Solution: We can abstract this problem as a graph with $n+1$ node. If monkey i is grasping monkey j, then we add an edge between node i and j. Hence, initially, this graph only has one connected component (indicating $m \geq n$), which means that all of the monkeys are hung directly or indirectly in the branch (denoted by 0). For each pair (u_t, v_t) we want to delete, checking whether monkey u_t or v_t will fall is equivalent to check whether u_t or v_t is still in the same connected component with node 0 (representing branch) after edge (u_t, v_t) is deleted. If we do search algorithm (BFS or DFS) here, this will require $2T \cdot O(m) = O(m^2)$ time complexity.

Now we explore the possibility of using the Union-Find data structure to improve the running time. The key problem is that Union-Find only supports "adding edges" (i.e. merging two sets) instead of "deleting edges". To resolve this problem, we consider the reversed process – we add those T edges back in a reverse order to decide the time some monkey falls. In more details, we first construct $n+1$ sets (each set only contains one node) (Step 2 of the algorithm description). Then, we add the remaining $m-T$ edges to this Union-Find data structure (Step 4 to Step 9). This gives us the scenario after time T, i.e. when all the T edges are deleted. Next, we add back those T in a reverse order to decide the time when each monkey falls (Step 10 to Step 23). Whenever the addition of an edge lets a monkey join the connected component where node 0 (i.e. the tree branch) is in, this is exactly the time when the monkey falls in the original process.

Before describing the complete algorithm, we need to modify the Union operation a little bit. Since we need to know the set of monkeys in the connected component that is to be merged with the node-0 component, it is convenient (and also efficient) to set up a linked list containing all the nodes in every connected component. We use a map $CC[]$ so that when u is the representative of its connected component, $CC[u]$ points to a linked list containing all the elements in the component. Initially each u is the representative of its own connected component, so $CC[u]$ points to a linked list with only one element u.

Whenever $Union(u, v)$ is called, we append the list $CC[u]$ to $CC[v]$ if v becomes the new representative, and append $CC[v]$ to $CC[u]$ otherwise. If we store both the head and tail pointers of the linked lists, this appending operation takes only $O(1)$ time, and does not affect the running time analysis of the Union-Find data structure.

With this modification, we describe our main algorithm as in Algorithm 1.

Proof of Correctness: Initially, all the monkeys haven't been assigned any time except for those monkeys in the same component with branch (node 0). When add back some edge (u_t, v_t), if both u_t and v_t are not in the same component with 0 (branch), it means that for original procedure deleting this edge will not lead some monkey to fall. Hence, in the algorithm, we just jump to Step 22. Otherwise, at least one of u_t and v_t is in the same component with 0. Hence, adding back this edge will lead both u_t and v_t to be in the same component with 0. If $FT(u_t) = 0$ (or $FT(v_t) = 0$), it means this is the first time that all the nodes in the same component with u_t (v_t respectively) to join the node-0 component. In the original procedure, deleting this edge will lead to the fall of all the nodes in the same component with u_t (v_t respectively).

Time Complexity:

We first omit those operations on Union-Find data structure and analyze them later. It is a little tricky to analyze the running time for Step 19 and Step 21. Note that one node will not be visited again whenever this node has been visited. Since there are $n+1$ nodes. Hence, the total running time for Step 19 and Step 21 is $O(n)$. Then, it is easy to see that to run the left procedures it requires $O(m)$ time noting that $T \leq m$ and $n \leq m$.

Next, we analyze those operations on Union-Find data structure. For Step 2, we are doing

Algorithm 1

1: **Input:** $(u_i, v_i), i = 1, \ldots, m$, $(u_t, v_t), t = 1, \ldots, T$
2: MakeUnionFind($\{0, 1, 2, \ldots, n\}$) and Initialize $CC[u]$ for each u
3: Build a hash function H and let $H((u_t, v_t)) = 1$, for $t = 1, 2, \ldots, T$
4: **for** $i \leftarrow 1$ **to** m **do**
5: **if** $H((u_i, v_i))$ does not exist **then**
6: $r_1 \leftarrow$ Find(u_i)
7: $r_2 \leftarrow$ Find(v_i)
8: **if** $r_1 \neq r_2$ **then**
9: Union(r_1, r_2)
10: $FT(j) \leftarrow 0$, for $j = 1, 2, \ldots, n$ ▷ $FT(j) = 0$ indicates that monkey j hasn't been assigned any time
11: $r_0 \leftarrow Find(0)$
12: For each node u in the same component with r_0, assign -1 to $FT(u)$ ▷ $FT(u) = -1$ indicates that monkey u will not fall after all T seconds
13: **for** $t \leftarrow T$ **downto** 1 **do**
14: $r_1 \leftarrow$ Find(u_t)
15: $r_2 \leftarrow$ Find(v_t)
16: $r_0 \leftarrow$ Find(0)
17: **if** $r_1 \neq r_0$ **and** $r_2 == r_0$ **then**
18: For each node u in $CC[r_1]$, assign t to $FT(u)$
19: **else if** $r_1 == r_0$ **and** $r_2 \neq r_0$ **then**
20: For each node v in $CC[r_2]$, assign t to $FT(v)$
21: **if** $r_1 \neq r_2$ **then**
22: Union(r_1, r_2)
23: **Output:** FT

initialization. The time complexity is $O(n)$. For Step 4 to Step 9, there are $2m$ Find operations. For Step 10 to Step 23, there are $3T + 1$ Find operations. Totally, since there are at most $n + 1$ sets, there are at most n Union operations. To sum up, there are at most $2m + 3T + 1 \leq 6m$ Find operations and n Union operations. Thus for these operations on Union-Find data structure, the running time is $O(6m\alpha(n)) + O(n) = O(m\alpha(n))$.

Therefore, the total running time is $O(m\alpha(n)) + O(m) = O(m\alpha(n))$.

Problem 3 [10 pts]

Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution: Suppose by way of contradiction that T and T' are two distinct minimum spanning trees of G. Since T and T' have the same number of edges, but are not of equal weight there is some edge $e' \in T'$ such that $e' \notin T$. If we add e' to T we get a cycle C. Let e be the most expensive edge on this cycle. Then by the Cycle Property, e does not belong to any minimum spanning tree,

contradicting the fact that it is at least one of T or T'.

Problem 4 [10 pts]

Let us say that a graph $G = (V, E)$ is a near tree if it is connected and has $n + 8$ edges where $n = |V|$. Give an algorithm with running time $O(n)$ that takes a near tree G with costs on its edges and returns a minimum spanning tree of G. Prove the correctness and running time of your algorithm. You may assume that all edge costs are distinct.

Solution: To do this we apply the Cycle Property 9 times. That is, we perform BFS until we find a cycle in the graph G, and then we delete the heaviest edge on this cycle. We have now reduced the number of edges in G by one, while keeping G connected, and by the Cycle property not changing the identity of the minimum spanning tree. We can simply, repeat this process 9 times. The running time would 9 times that of BFS, which is $9 \cdot O(n) = O(n)$.

Problem 5 [12 pts]

For each of the following statements decide if it is TRUE or FALSE. If it is TRUE, give a short explanation. If it is FALSE, give a counterexample.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be the Minimum Spanning Tree for this instance. Now suppose we replace each edge cost c_e by its square c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

TRUE or FALSE? T must still be a Minimum Spanning Tree for the new instance.

(b) Suppose were given an instance of the Shortest s-t Path Problem on a directed graph G. We assume that all edge costs are positive and distinct. Let P be a minimum cost s-t path for this instance. Now suppose we replace each edge cost c_e by its square c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

TRUE or FALSE? P must still be a minimum cost s-t path for the new instance.

Solution:

(a) True. If we feed the costs c_e^2 into Kruskal's algorithm, it will sort them in the same order, hence put the same subset of edges in the MST.

(b) False. Let G have edges (s, v), (v, t) and (s, t), where the first two of these edges have costs 2 and 4, and third has cost 5. Then the shortest path is the single edge (s, t), but after squaring the costs would go through v.

Problem 6 [10 pts]

Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G. Now assume that a new edge is added to G, connecting two nodes $v, w \in V$ with cost c.

(a) Give an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumptions you make about what data structure is used to represent the tree T and the graph G.

(b) Suppose T is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree T to the new minimum-cost spanning tree.

Solution:

(a) Let $e = (v, w)$ be the new edge e being added. We represent T using an adjacency list, and we find the v-w path P in T in time linear in the number of nodes and edges of T, which is $O(|V|)$. If every node in this path in T has cost less than c, then the cycle property implies that the new edge $e = (v, w)$ is not in the minimum spanning tree, since it is the most expensive edge on the cycle C formed from P and e, so the minimum spanning tree has not changed. On the other hand, if some edge on this path has cost greater than c, then the Cycle Property implies that the most expensive such edge f cannot be in the minimum spanning tree, and T is no longer the minimum spanning tree.

(b) We replace the heaviest edge on the v-w path P in T with the edge $e = (v, w)$, obtaining a new spanning tree T'. We claim that T' is a minimum spanning tree. To prove this consider any edge e' not in T', and show that we can apply the Cycle Property to conclude that e' is not in any MST. So let $e' = (v', w')$. Adding e' to T' gives us a cycle C' consisiting of the v'-w' path P' in T', plus e'. If we can show e' is the most expensive edge we are done.
To do this we consider one further cycle: the cycle K formed by adding e' to T. By the Cycle Property, e' is the most expensive edge on K. Edge f is the most expensive edge on C, edge e' is the most expensive edge on K. Now if the new edge e does not belong to C', then $C' = K$, and so e' is the most expensive edge on C'. Otherwise, the cycle K includes f (since C' needed to use e instead), and C' uses a portion of C (including e) and a portion of K. In this case, e' is more expensive than f (since f lies on K), and hence it is more expensive than everything on C (since f is the most expensive edge on C). It is also more expensive than everything else on K, and so it is the most expensive edge on C' as desired.
Now if the edge costs are not distinct, we first perturb all the edge costs by a very small amount so they become distinct. Moreover, we do this so we add a very small quantity ϵ to the new edge e, and we perturb the costs of all other edges f by even much smaller distinct quantities δ_f. For a tree T, let $c(T)$ denote its real cost and $c'(T)$ denote its perturbed cost. Now we use the above solution with distinct edge costs. Our perturbation has the following two properties.

(a) First, for trees T_1 and T_2, if $c'(T_2) < c'(T_1)$, then $c(T_2) \leq c(T_1)$.

(b) Second, if $c(T_1) = c(T_2)$, and T_2 contains e but T_1 does not then $c(T_2) > c(T_1)$.

It follows from these two properties that our conclusion in (a) is correct: since $c'(T') < c'(T_1)$, and T' contains T' contains e but T doesn't, property (i) implies $c(T') \leq c'(T)$, and property (ii) implies $c(T') < c'(T)$. In (b), we compute a MST with respect to the perturbed costs which by property (i), is aso one the MSTs with respect to the real costs.

Problem 7 [12 pts]

Your friends are planning an expedition to a small town deep in the Canadian north next winter break. Theyve researched all the travel options and have drawn up a directed graph whose nodes represent intermediate destinations and edges represent the roads between them.

In the course of this, theyve also learned that extreme weather causes roads in this part of the world to become quite slow in the winter and may cause large travel delays. Theyve found an excellent travel Web site that can accurately predict how fast theyll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the Web site answers queries of the following form: given an edge $e = (v, w)$ connecting two sites v and w, and given a proposed starting time t from location v, the site will return a value $f_e(t)$, the predicted arrival time at w. The Web site guarantees that $f_e(t) \geq t$ for all edges e and all times t (you cant travel backward in time), and that $f_e(t)$ is a monotone increasing function of t (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + \ell_e$, where ℓ_e is the time needed to travel from the beginning to the end of edge e.

Your friends want to use the Web site to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the Web site are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the Web site (based on a specific edge e and a time t) as taking a single computational step.

Solution: The following modification of the Dijkstra's algorithm solves the problem.

Algorithm 2

1: Let S be the set of explored nodes.
2: For each $u \in S$, we store the earliest time $d(u)$ when we can arrive at u and the last site $r(u)$ before u on the fastest path to u
3: Initially $S = \{s\}$ and $d(s) = 0$
4: **while** $S \neq V$ **do**
5: Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} f_e(d(u))$ is as small as possible.
6: Add v to S and define $d(v) = d'(v)$ and $r(v) = u$

To establish the correctness of the algorithm, we will prove by induction on the size of S that for all $v \in S$, the value $d(v)$ is the earliest time that water will reach v.
The case $|S| = 1$ is clear, because we know that water starts from the node s at time 0. Suppose this claim holds for $|S| = k \geq 1$; we now grow S to size $k+1$ by adding the node v. For each $x \in S$, let P_x be a path taken by water to reach x at time $d(x)$. Let (u, v) be the final edge on the path P_v from s to v. Consider any other s-v path P. We will show that the time at which v is reached

using P is at least $d(v)$. In order to reach v, this path must leave the set somewhere; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y. Let P' be the sub-path of P up to node y. By the choice of node v in iteration $k+1$, the travel time to y using P' is at least as large as the travel time to v using P_v. Since the time varying edges do not allow traveling back in time, this means that the route to v through y is at least $d(v)$, as desired.

Problem 8 [12 pts]

A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge e is a communication link, with a given available bandwidth b_e.

For each pair of nodes $u, v \in V$, they want to select a single $u - v$ path P on which this pair will communicate. The *bottleneck rate* $b(P)$ of this path P is the minimum bandwidth of any edge it contains; that is, $b(P) = \min_{e \in P} b_e$. The *best achievable bottleneck rate* for the pair u, v in G is simply the maximum, over all $u - v$ paths P in G, of the value $b(P)$.

Its getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: Maybe one can find a spanning tree T of G so that for *every* pair of nodes u, v, the unique $u - v$ path in the tree actually attains the best achievable bottleneck rate for u, v in G. (In other words, even if you could choose any $u - v$ path in the whole graph, you couldnt do better than the $u - v$ path in T.)

This idea is roundly heckled in the offices of CluNet for a few days, and theres a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.

Show that such a tree exists, and give an efficient algorithm to find one. That is, give an algorithm constructing a spanning tree T in which, for each $u, v \in V$, the bottleneck rate of the $u - v$ path in T is equal to the best achievable bottleneck rate for the pair u, v in G.

Solution: We claim that a minimum spanning tree, computed with each edge cost equal to the negative of its bandwidth has this property; and so it is enough to compute a minmum spanning tree. We first prove this with the assumption that all edge costs are distinct and then show that it remains true even without this assumption. Suppose the claim is not true. then there is some pair of nodes u,v for whcihc the path P in the MST does not have a bottleneck rate as high as some other u-v path P'. Let $e = (x, y)$ be an edge of minimum bandwidth on the path P; note that $\notin P'$. Moreover, e has the smallest bandwidth of any edge in $P \cup P'$. Now using the edges in $P \cup P'$ other than e, it is possible to travel from x to y. Thus there is a simple path from x to y using these edges, and so there is a cycle C on which e has the minimum bandwidth. This means that in our minimum spanning tree instance, e has the maximum cost on the cycle C; but e belongs to the MST, contradicting the cycle property.

Now, if the edge costs are not all distinct, we perturb all the edge bandwidths by extremely small amounts so they become distinct, and then find a MST. We refer for each edge e, to real bandwidth as b_e and perturbed bandwidth b'_e. In particular, we choose perturbations small enough so that if $b_e > b_f$ for edges e and f, then also $b'_e > b'_f$. Our tree has the best bottleneck rate for all pairs, under the perturbed bandwidths. But suppose that the u-v path P in this tree did not have the

best bottleneck rate if we consider the original bandwidths; say there is a better path P'. Then there is an edge e on P for which $b_e > b_f$ for all edges f on P'. But the perturbations were so small that they did not cause any edges with with distinct bandwidths to change the relative order of their bandwidth values, so it would follow that $b'_e > b'_f$ for edges f on P', contradicting our conclusion that P had the best bottleneck rate with respect to the perturbed edges.

Problem 9 [10 pts]

You have n people who can work at a store, where person i will work from time b_i until time e_i. Also, you are given an opening time b and closing time e for the store. The goal is to find the minimum number of people to work for the day so that at least one person is at the store between times b and e. You can assume that for all i, $b_i \geq b$ and $e_i \leq e$.

Give a greedy algorithm that optimally solves this problem.

Solution: We start by selecting people iteratively. Let P denote the set of people remained to be selected, and S be the set of people already selected. Here is the algorithm.

Algorithm 3

Input: $b_i, e_i, i = 1, \ldots, n$, b and e
$P = \{1, 2, 3, \ldots, n\}$, $S = \emptyset$, $e' \leftarrow b$
while $e' < e$ **do**
    $i^* = \text{argmax}_{i \in P}\{e_i | b_i \leq e'\}$
    if $e_{i^*} > e'$ **then**
        $P \leftarrow P \backslash \{i^*\}$
        $S \leftarrow S \cup \{i^*\}$
        $e' \leftarrow e_{i^*}$
    else
        Output: No solution
Output: The set S

Proof of Correctness: If the algorithm outputs "No solution", it means no solution will be found. At the time this algorithm stops, suppose P, S and e' become P_t, S_t and e'_t respectively. Then we know that the whole set $\{1, 2, 3, \ldots, n\}$ can be separated into two disjoint sets $A = S_t \cup \{i | i \in P \text{ and } b_i \leq e'_t\}$ and $B = P_t \backslash \{i | i \in P \text{ and } b_i \leq e'_t\}$ satisfying $\max\{e_i | i \in A\} = e'_t < \min\{b_i | i \in B\}$ (this value equals to e when B is empty). Hence, there are no persons in P covering time $(e'_t, \min\{b_i | i \in B\})$. Therefore, for this instance, there are no solutions.

If not, it is trivial that S is a feasible solution. Suppose the first person selected by this algorithm in S is i_1. We have the following lemma.

Lemma 1. *There exists an optimal solution S' such that $i_1 \in S'$.*

Proof. If not, suppose one possible optimal solution is S^* satisfying $i_1 \notin S^*$. Since S^* is a solution, then $\{e_i | b_i \leq b \text{ and } i \in S^*\}$ will not be empty. By the selection of i_1, we know $e_{i_1} \geq \max\{e_i | b_i \leq b \text{ and } i \in S^*\}$. Then, let $S' = S^* \backslash \{e_i | b_i \leq b \text{ and } i \in S^*\} \cup \{i_1\}$. It will also be a solution satisfying $|S'| \leq |S^*|$. Since S^* is optimal, S' will also be optimal. So far, we have constructed an optimal solution including i_1. □

After proving this lemma, the left will be simple. We can further prove that there always exists one optimal solution including the persons we have already selected.

Time Complexity: It suffices to only consider the recursive part. There are at most n iterations since for each iteration, at least one person will be selected. And for each iteration, the selection for i^* requires $O(n)$ time. So totally, time complexity is $O(n^2)$.