1. **(a)** Consider two soldiers with $r_1 = 1$, $c_1 = 12$, $r_2 = 10$, and $c_2 = 1$. With the greedy algorithm, the second solider will wait for the first soldier to climb and can start run at time 13. The total finish time is therefore 23. If we reverse the order of the two soldier, the finish time is instead 14.

   (b) Sort the soldiers so that $r_i \geqslant r_{i+1}$.

   (c) [*Intuition*] The exchange argument is to show that for any i, if we schedule soldier i before $i+1$ and $r_i < r_{i+1}$ then we can reverse this order and obtain a solution that is at least as good as before the soldiers were interchanged. Note that when we exchange two soldiers the finishing time for the previous and remaining soldiers is not changed. Before the exchange, soldier $i+1$ finishes at time $\left(\sum_{j=1}^{i-1} c_j\right) + c_i + c_{i+1} + r_{i+1}$. After the exchange, soldier i finishes at time $\left(\sum_{j=1}^{i-1} c_j\right) + c_i + c_{i+1} + r_i$ and soldier $i+1$ finishes at time $\left(\sum_{j=1}^{i-1} c_j\right) + c_{i+1} + r_{i+1}$. Both are less than the finish time of soldier $i+1$ before the exchange. Iteratively apply this exchange argument to prove that the greedy algorithm returns the optimal solution.

   (c) [*Formal structure*] Let $S = [(r_1, c_1), \ldots, (r_n, c_n)]$ be the order produced by our algorithm. We say that the partial sequence $S_k = [(r_1, c_1), \ldots, (r_k, c_k)]$ is "promising" iff there is some optimum sequence $S^* = [(r_1^*, c_1^*), \ldots, (r_n^*, c_n^*)]$ such that $(r_1^*, c_1^*) = (r_1, c_1), \ldots, (r_k^*, c_k^*) = (r_k, c_k)$—note that S^* is optimum means that its overall finish time $\max(c_1^* + r_1^*, c_1^* + c_2^* + r_2^*, \ldots, c_1^* + c_2^* + \cdots + c_n^* + r_n^*)$ is as small as possible. We prove that S_k is promising by induction on k.

   Base Case: $S_0 = ()$ is promising because any optimum solution extends it.

   Ind. Hyp.: Suppose that $k \geqslant 0$ and S_k is promising, with optimum solution S^* extending it (as defined above).

   Ind. Step: We prove that there is an optimum solution S' that extends $S_{k+1} = [S_k, (r_{k+1}, c_{k+1})]$.

   Case 1: If $(r_{k+1}^*, c_{k+1}^*) = (r_{k+1}, c_{k+1})$, then S^* already extends S_{k+1}.

   Case 2: If $(r_{k+1}^*, c_{k+1}^*) \neq (r_{k+1}, c_{k+1})$, then let $j > k+1$ be the index such that $(r_j^*, c_j^*) = (r_{k+1}, c_{k+1})$. Let $S' = [(r_1^*, c_1^*), \ldots, (r_k^*, c_k^*), (r_j^*, c_j^*), (r_{k+2}^*, c_{k+2}^*), \ldots, (r_{j-1}^*, c_{j-1}^*), (r_{k+1}^*, c_{k+1}^*), (r_{j+1}^*, c_{j+1}^*), \ldots, (r_n^*, c_n^*)]$, i.e., the same as S^* but with soldiers $k+1$ and j swapped. For convenience, let $t_i = c_1 + c_2 + \cdots + c_i + r_i$ be the finish time of soldier i in S, $t_i^* = c_1^* + c_2^* + \cdots + c_i^* + r_i^*$ be the corresponding time for S^*, and. $t_i' = c_1' + c_2' + \cdots + c_i' + r_i'$ be the corresponding time for S'. Then:

   - $t_1' = t_1^* = t_1, \ldots, t_k' = t_k^* = t_k$ (by definition of S^* and S').
   - $t_{k+1}' = t_{k+1} = c_1 + \cdots + c_k + c_{k+1} + r_{k+1} \leqslant c_1 + \cdots + c_k + c_{k+1}^* + \cdots + c_{j-1}^* + c_{k+1} + r_{k+1} = t_j^*$ (because $c_j^* = c_{k+1}$ and $r_j^* = r_{k+1}$).
   - $$\begin{aligned} t_{k+2}' &= c_1 + \cdots + c_k + c_{k+1} + c_{k+2}^* + r_{k+2}^* \\ &\leqslant c_1 + \cdots + c_k + c_{k+1} + c_{k+2}^* + \cdots + c_{j-1}^* + r_{k+2}^* \\ &\leqslant c_1 + \cdots + c_k + c_{k+1}^* + c_{k+2}^* + \cdots + c_{j-1}^* + c_{k+1} + r_{k+2}^* \\ &\leqslant c_1 + \cdots + c_k + c_{k+1}^* + c_{k+2}^* + \cdots + c_{j-1}^* + c_{k+1} + r_{k+1} \quad = t_j^* \end{aligned}$$
   - Similarly, $t_{k+3}' \leqslant t_j^*, \ldots, t_{j-1}' \leqslant t_j^*$.
   - Finally, $t_j' = t_j^*, \ldots, t_n' = t_n^*$.

   Hence, $\max(t_1', \ldots, t_n') \leqslant \max(t_1^*, \ldots, t_n^*)$; in other words, S' is an optimum solution that extends S_{k+1}.

   In both cases, S_{k+1} is promising.

   Therefore, every partial solution is promising, so the greedy solution is optimum.

2. **Algorithm:** *Pseudo-code*

```
$s_0 \leftarrow 0$
$s_i \leftarrow \sum_{x=1}^{i} a_x$
$s'_i \leftarrow \min_{0 \leqslant x \leqslant i} s_x$
$v_i \leftarrow \arg\min_{0 \leqslant x \leqslant i} s_x$   (i.e., $s'_i = s_{v_i}$)
$best \leftarrow a_1$
$j \leftarrow 1$
$k \leftarrow 1$
for $h \leftarrow 2, \dots, n$:
    if $best < s_h - s'_{h-1}$:
        $best \leftarrow s_h - s'_{h-1}$
        $j \leftarrow v_{h-1} + 1$
        $k \leftarrow h$
```

Correctness Proof:

- Note that $s_k - s_{j-1}$ is the sum of the subsequence $a_j, \dots, a_k$, because:

$$s_k - s_{j-1} = \left(\sum_{x=1}^{k} a_x\right) - \left(\sum_{x=1}^{j-1} a_x\right) = \sum_{x=j}^{k} a_x$$

- $a_{v_{k-1}+1}, \dots, a_k$ is the maximum subsequence that ends at index k. This is because for any other subsequence $a_j, \dots, a_k$, we have:

$$a_{v_{k-1}+1} + \cdots + a_k = s_k - s_{v_{k-1}} \geqslant s_k - s_j = a_{j+1} + \cdots + a_k$$

- *Proof with Induction*: At each iteration, *best* is equal to the sum of the maximum subsequence with an end index in the range $1, \dots, h$; j and k corresponds to the start index and the end index of that subsequence.
  Base: When $h = 1$, the only subsequence is the one with a_1. It holds trivially based on the initial values of *best*, j, and k.
  Induction Step: From the induction hypothesis, we know that before the iteration h, *best*, j, and k correspond to best sequence that ends with indexes $1, \dots, h-1$. By the observations above, the algorithm will update these variables accordingly if the best subsequence that ends with h, i.e., $a_{v_{k-1}+1}, \dots, a_k$, has a greater sum. Therefore after the iteration, *best*, j, and k now correspond to the best subsequence among all subsequences that end with indexes in the range $1, \dots, h$.

Alternative Algorithm:

```
MAXSUM($a_1, a_2, \dots, a_n$):
    $m \leftarrow \ell \leftarrow a_1$
    $h \leftarrow i \leftarrow j \leftarrow k \leftarrow 1$
    Loop Invariant: $1 \leqslant h \leqslant i \leqslant n$; $1 \leqslant j \leqslant k \leqslant i$; $\ell = a_h + a_{h+1} + \cdots + a_i$ is the maximum sum of
        any subsequence whose last element is $a_i$; $m = a_j + a_{j+1} + \cdots + a_k$ is the maximum sum
        of any non-empty subsequence of $a_1, a_2, \dots, a_i$.
    for $i \leftarrow 2, 3, \dots, n$:
        if $\ell < 0$:
            $\ell \leftarrow a_i$
            $h \leftarrow i$
        else:
            $\ell \leftarrow \ell + a_i$
        if $\ell > m$:
            $m \leftarrow \ell$
            $(j,k) \leftarrow (h,i)$
    return $(j,k)$
```

Correctness is based on the same argument as for the other algorithm (the principle is the same, it's just computed differently).

3. **Algorithm** *Pseudo-code*

```
Compute the minimum spanning tree $T^*$ of $G = (V,E)$
Sort all edges that are not in $T^*$ such that $w(e_1) \leqslant w(e_2) \leqslant \cdots \leqslant w(e_m)$
$best \leftarrow \infty$
$T \leftarrow \varnothing$
for $i \leftarrow 1,\ldots,m$:
    Suppose $e_i$ connects $u$ and $v$
    Let $P$ be the path from $u$ to $v$ in $T^*$
    Let $e'$ be the edge with maximum weight in $P$
    if $best > w(T^*) + w(e_i) - w(e')$:
        $best \leftarrow w(T^*) + w(e_i) - w(e')$
        $T \leftarrow (T^* - \{e'\}) \cup \{e_i\}$
```

Correctness Proof: Note that the algorithm starts from the minimum spanning tree T^*, adds an additional edge x into the tree, and then removes the edge y with the maximum weight in the cycle of $T^* \cup \{x\}$. The algorithm returns $T = T^* \cup \{x^*\} - \{y^*\}$ as the second minimum tree such that:

$$w(x^*) - w(y^*) \leqslant w(x) - w(y)$$

In other words, T has smaller weight than any other tree obtained from adding and removing one edge from T^*.

Now consider any spanning tree T' obtained by adding and removing more than one edge from T^*, i.e., $T' = T^* \cup \{x_1,\ldots,x_k\} - \{y_1,\ldots,y_k\}$ and $k > 1$. Note that for any $1 \leqslant j \leqslant k$ we have $w(x_j) - w(y_j) \geqslant 0$, because otherwise $(T^* \cup \{x_j\}) - \{y_j\}$ would have a smaller total weight than T^* and this contradicts with the fact that T^* is the minimum spanning tree. We have:

$$w(T') = w(T^*) + \sum_{j=1}^{k}(w(x_j) - w(y_j)) \geqslant w(T^*) + w(x_1) - w(y_1) \geqslant w(T^*) + w(x^*) - w(y^*) = w(T)$$

T has a smaller or equal weight sum than T'. Therefore T has smaller weights than any other spanning tree except T^*, i.e., the returned result T is the second minimum spanning tree.

Optimization and Complexity: The naive implementation of the algorithm would have time complexity $\mathcal{O}(|E||V|)$, because each iteration needs to go over all edges in the cycle between u and v and there could be $\mathcal{O}(|V|)$ edges in the worst scenario.

To this end, we maintain a table $f(v,k)$ for each node v, where $0 \leqslant k \leqslant \log|V|$. $f(v,k) = \langle v', e' \rangle$, where v' corresponds to the 2^k-th ancestor of v in T^* and e' is the edge with the maximum weight in the path from v to v' in T^*. This table can be computed in $\mathcal{O}(|V|\log|V|)$ time. With the help of this table, we can find the least common ancestor of u and v in each iteration of the algorithm within $\mathcal{O}(\log|V|)$. Suppose the common ancestor is a, we can also find the edge with the maximum weight in paths (u, a) and (a, v) in $\mathcal{O}(\log|V|)$. Therefore the algorithm now runs in $\mathcal{O}(|E|\log|V|)$ after the minimum spanning tree step.