Name: Ananye Agarwal **Entry Number:** 2017CS10326
Name: Kabir Tomer **Entry Number:** 2017CS50410
Name: Rajat Jaiswal **Entry Number:** 2017CS50415

1. (12 points) Given a strongly connected directed graph, $G = (V, E)$ with positive edge weights along with a particular node $v \in V$. You wish to pre-process the graph so that queries of the form "what is the length of the shortest path from s to t that goes through v" can be answered in constant time for any pair of distinct vertices s and t. The pre-processing should take the same asymptotic run-time as Dijkstra's algorithm. Analyse the runtime and provide a proof of correctness.

Solution: The condition imposed in the problem is that we have to visit v while travelling from s to t. In order to do so, we will have to first visit v from s and then from v we have to visit t. Dijkstra's algorithm will give us the shortest path length(property discussed in the class).

Algorithm 1 Shortest path between two nodes that go through v

```
1:
2: Allocate array from[] of size |V| //Stores shortest distance to all the vertices from the vertex v
3: Allocate array to[] of size |V|    //Stores shortest distance from all the vertices to the vertex v
4:
5: function PREPROCESSING-STEP(G, v)
6:     G^R ← Reverse(G)
7:     from[] ← Dijkstra(G, v)
8:     to[] ← Dijkstra(G^R, v)
9: end function
10:
11: function QUERY-SHORTESTPATH(s, t)
12:     return from[t] + to[s]
13: end function
14:
```

Running time analysis: Reversing the graph in line 6 takes $O(|V|+|E|)$ time, which is linear. Line $7, 8$ each takes the same amount of time as to run the Dijsktra's algorithm. So, the overall running time of the preprocessing step is asymptotically same as Dijsktra's algorithm. Depending on the implementation of the priority queue in Dijsktra's algorithm, the running time of the preprocessing step can be $O(|V|^2)$[Linked list] or $O((|V| + |E|) \log |V|)$[Binary heap] or $O((d|V| + |E|) \log_d |V|)$[$d$-ary heap] or $O((|E| + |V| \log |V|)$[Fibonacci heap]. Answering a query take $O(1)$ time as it just involves fetching two elements of arrays.

Proof of Correctness: The length of the shortest path from s to t that goes through v is equal to length of shortest path from s to v plus the length of shortest path from v to t. If a shorter path than this is found then either there exists a path from v to t whose length is smaller than $from[t]$, or there exists a path from s to v whose length is smaller than $to[s]$. In either case, contradiction. During preprocessing we precisely compute these values which enables us to answer the query in constant time.

- We run Dijkstra's algorithm on G with starting node v that returns an array of the shortest path from v to all the vertices in the array $from[.]$. This will give us the shortest distance between v and t denoted by $from[t]$.
- And also run Dijkstra's algorithm on G^R(the reverse-graph of G) with starting node v that returns an array of the shortest path from all the vertices to v. The length of the shortest path from a to b in G is also the length of the shortest path from b to a in G^R, as the direction of edges have now been reversed so will be the path. This gives us the shortest distance between v and s in G^R denoted by $to[s]$. This is same as shortest distance between s and v in G.

2. Counterexamples are effective in ruling out certain algorithmic ideas. In this problem, we will see a few such cases.

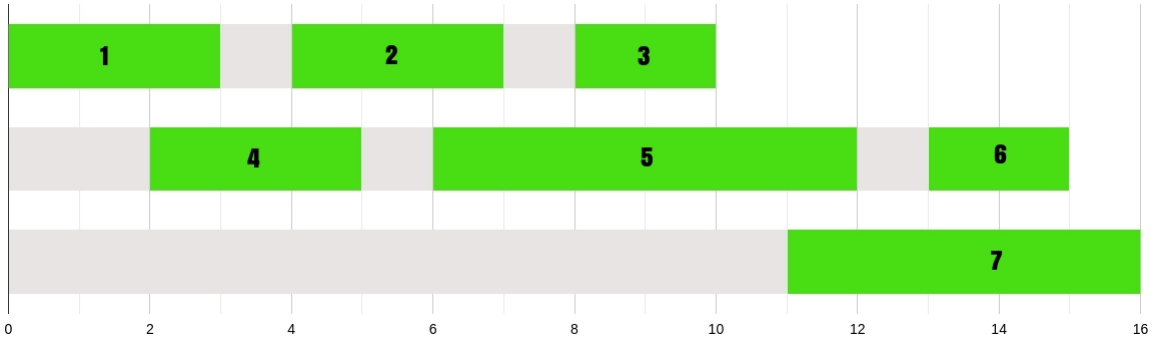
   (a) (5 points) Recall the following event scheduling problem discussed in class (lecture 15):

   > You have a conference to plan with n events and you have an unlimited supply of rooms. Design an algorithm to assign events to rooms in such a way as to minimize the number of rooms.

   The following algorithm was suggested during class discussion. `ReduceToSingleRoom`$(E_1, ..., E_n)$
   - $U \leftarrow \{E_1, ..., E_n\}$; $i \leftarrow 1$
   - While U is not empty:
     - Use Earliest Finish Time greedy algorithm on events in set U to schedule a subset $T \subseteq U$ of events in room i
     - $i \leftarrow i + 1$; $U \leftarrow U \setminus T$

   Show that the above algorithm does not always return an optimal solution.

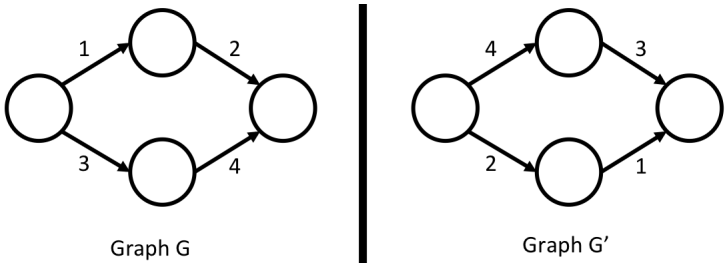
   Solution: Proof by counterexample. Consider the event schedule given below. The given algorithm gives minimum number of rooms as 3, namely, $\text{Room}_1 = \{ 1, 2, 3, 6 \}$, $\text{Room}_2 = \{ 4, 5 \}$, $\text{Room}_3 = \{ 7 \}$. But the optimal solution is 2 rooms, which are $\text{Room}_1 = \{ 1, 2, 3, 7 \}$, $\text{Room}_2 = \{ 4, 5, 6 \}$.



   Event Schedule

   (b) (5 points) A longest simple path from a node s to t in a weighted, directed graph is a simple path from s to t such that the sum of weights of edges in the path is maximised. Here is an idea for finding a longest path from a given node s to t in any weighted, directed graph $G = (V, E)$:

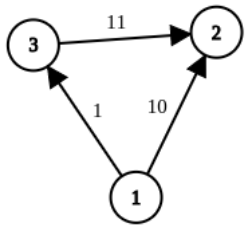
   > Let the weight of the edge $e \in E$ be denoted by $w(e)$ and let w_{max} be the weight of the maximum weight edge in G. Let G' be a graph that has the same vertices and edges as G but for every edge $e \in E$, the weight of the edge is $(w_{max} + 1 - w(e))$. (*For example, consider the graph G below and its corresponding graph G'.*)



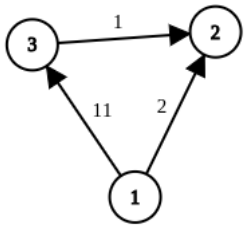
   > Run Dijkstra's algorithm on G' with starting vertex s and return the shortest path from s to t.

   Show that the above algorithm does not necessarily output the longest simple path.

Solution: Proof by counterexample. Consider the Graph G below. The corresponding Graph G' has also been shown. The longest simplest path from vertex 1 to 2 should be the path $1 \to 3 \to 2$. But the proposed algorithm would return the path $1 \to 2$, which is incorrect.



Graph G



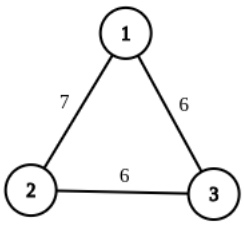
Graph G'

(c) (5 points) Recall that a *Spanning Tree* of a given connected, weighted, undirected graph $G = (V, E)$ is a graph $G' = (V, E')$ with $E' \subseteq E$ such that G' is a tree. The cost of a spanning tree is defined to be the sum of weight of its edges. A *Minimum Spanning Tree (MST)* of a given connected, weighted, undirected graph is a spanning tree with minimum cost. The following idea was suggested for finding an MST for a given graph in the class.

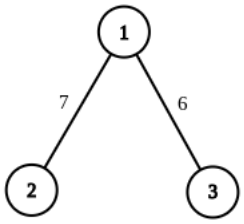
> Dijkstra's algorithm gives a shortest path tree rooted at a starting node s. Note that a shortest path tree is also a spanning tree. So, simply use Dijkstra's algorithm and return the shortest path tree.

Show that the above algorithm does not necessarily output a MST. In other words, a shortest path tree may not necessarily be a MST. (*For this question, you may consider only graphs with positive edge weights.*)

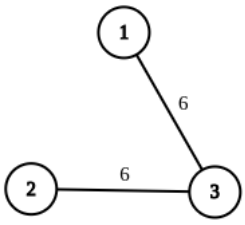
Solution: Proof by counterexample. Consider the Graph G below. The MST of the graph M has also been shown. The spanning tree D that Dijsktra returns is not the spanning tree with the least cost. Cost of M is 12, whereas the the cost of the D is 13. Hence, the proposed algorithm is incorrect.



Graph G



Spanning Tree D



MST M

3. (*Example for "greedy stays ahead"*) Suppose you are placing sensors on a one-dimensional road. You have identified n possible locations for sensors, at distances $d_1 \leq d_2 \leq ... \leq d_n$ from the start of the road, with $0 \leq d_1 \leq M$ and $d_{i+1} - d_i \leq M$. You must place a sensor within M of the start of the road, and place each sensor after that within M of the previous one. The last sensor must be within M of d_n. Given that, you want to minimize the number of sensors used. The following greedy algorithm, which places each sensor as far as possible from the previous one, will return a list $d_{i_1} \leq d_{i_2} \leq ... \leq d_{i_k}$ of locations where sensors can be placed.

`GreedySensorMin(`$d_1 ... d_n, M$`)`
- Initialize an empty list
- Initialize $I = 1$, $PreviousSensor = 0$.

- While $(I < n)$:
  - While ($I < n$ and $d_{I+1} \leq PreviousSensor + M$) $I++$
  - If $(I < n)$ Append d_I to list; $PreviousSensor = d_I; I++$.
- if list is empty, append d_n to list
- return(list)

In using the "greedy stays ahead" proof technique to show that this is optimal, we would compare the greedy solution $d_{g_1}, ..d_{g_k}$ to another solution, $d_{j_1}, ..., d_{j_{k'}}$. We will show that the greedy solution "stays ahead" of the other solution at each step in the following sense:
Claim: For all $t \geq 1, g_t \geq j_t$.

(a) (5 points) Prove the above claim using induction on the step t. Show base case and induction step.

Solution: From problem definition we know that if $i \geq j$, then $d_i \geq d_j$.
Induction Hypothesis: $P(t) : g_t \geq j_t$ is true, For all $t \geq 1$.

Base Case: As per the problem formulation, both d_{g_1} and d_{j_1} has to within the distance M from 0. The way greedy algorithm has been designed, d_{g_1} is the last location along the road that is within distance M from 0. Therefore, $g_1 \geq j_1$.

Induction Step: Assume that $P(1)$, $P(2)$, $P(3)$, ..., $P(t-1)$ is true. We will show that $P(t)$ is true. For the sake of contradiction assume that $g_t < j_t$. From the problem formulation we know that $d_{j_t} - d_{j_{t-1}} \leq M$, and from induction hypothesis we know that $g_{t-1} \geq j_{t-1}$. This implies that $d_{j_t} - d_{g_{t-1}} \leq M$. We have assumed that $g_t < j_t$, hence in this scenario the greedy algorithm will not pick d_{g_t} but will pick d_{j_t} since d_{j_t} is farther than d_{g_t} and is also a feasible solution, which is a contradiction. Hence, $g_t \geq j_t$, and $P(t)$ is true.

(b) (3 points) Use the claim to argue that $k' \geq k$. (Note that this completes the proof of optimality of the greedy algorithm since it shows that greedy algorithm places at most as many sensors as any other solution.)

Solution: For the sake of contradiction assume that $k' < k$. It is given that $d_{j_1}, ..., d_{j_{k'}}$ is a valid solution, so from the problem formulation we have that $d_n - d_{j_{k'}} \leq M$. From part (a) we know that $g_{k'} \geq j_{k'}$. This implies that $d_n - d_{g_{k'}} \leq M$. Then this implies that the greedy algorithm will terminate at this point and will not place any more sensors. But the greedy solution is till d_{g_k} and $k > k'$. This is a contradiction owing to claim in part(a). Therefore, we establish that $k' \geq k$.

(c) (2 points) In big-O notation, how much time does the algorithm as written take? Write a brief explanation.

Solution: I goes from 1 to n, so the algorithm precisely iterates once over the list of sensors, making a single pass. Appending to the list takes $O(1)$ time and all other arithmetic and logical operations also take $O(1)$ time. Hence, the time is proportional to the size of the list. Therefore, the algorithm runs in $O(n)$ time.

4. (*Example for "modify the solution"*) You have n cell phone customers who live along a straight highway, at distances $D[1] < D[2] < ... < D[n]$ from the starting point. You need to have a cell tower within Δ distance of each customer, and want to minimize the number of cell towers.

(*For example, consider* $\Delta = 3$ *and there are* 3 *customers (i.e.,* $n = 3$*) with* $D[1] = 3, D[2] = 7, D[3] = 10$. In this case, you can set up two cell towers, one at 6 and one at 10.)

Here is a greedy strategy for this problem.

Greedy strategy: Set up a tower at distance d which is at the farthest edge of the connectivity range for the customer who is closest to the starting point. That is, $d = \Delta + D[1]$. Note that

all customers who are within Δ distance of this tower at d, are covered by this tower. Then recursively set up towers for the remaining customers (who are not covered by the first tower).

We will show that the above greedy strategy gives an optimal solution using modify-the-solution. For this, we will first need to prove the following exchange lemma.

Exchange Lemma: *Let G denote the greedy solution and let g_1 be the location of the first cell phone tower set up by the greedy algorithm. Let OS denote any solution that does not have cell phone tower at g_1. Then there exists a solution OS' that has a cell phone tower set up at g_1 and OS' has the same number of towers as OS.*

Proof. Let $OS = \{o_1, ..., o_k\}$. That is, the locations of the cell phone towers as per solution OS is $o_1 < o_2 < ... < o_k$. We ask you to complete the proof of the exchange lemma below.

(a) (1 point) Define OS'.

Solution: $OS' = (OS - \{o_1\}) \cup \{g_1\}$.

(b) (2 points) OS' is a valid solution because ... (*justify why OS' provides coverage to all customers.*)

Solution: OS' is a valid solution because $o_1 \leq g_1$. If $o_1 > g_1$ was true then o_1 would have not covered $D[1]$. g_1 covers $D[1]$ which is the left most customer and it also covers all the customers that were covered by o_1 because $o_1 \leq g_1$. Therefore, OS' is a valid solution.

(c) (2 points) The number of cell phone towers in OS' is at most the number of cell phone towers in OS because... (*justify*)

Solution: We have removed o_1 from OS and added g_1 to get OS'. Therefore the number of cell phone towers in OS' is at most the number of cell phone towers in OS. $|OS'| \leq |OS|$.

We will now use the above exchange lemma to argue that the greedy algorithm outputs an optimal solution for any input instance. We will show this using mathematical induction on the input size (i.e., number of customers). The base case for the argument is trivial since for $n = 1$, the greedy algorithm opens a single tower which is indeed optimal.

(a) (3 points) Show the inductive step of the argument.

Solution: Assume that the greedy algorithm outputs an optimal solution for all input instance with k customers such that $1 \leq k < n$. Now we will show that greedy algorithm outputs an optimal solution for any input instance I with n customers.

Let g_1 be the location of the first cell phone tower set up by the greedy algorithm in $GS(I)$ and S be the set of customers in I that are covered by this tower at g_1. And $I' = I \backslash S$ ($\backslash$ denotes set difference). Clearly, $|I'| < |I|$ because S contains atleast $D[1]$. Let OS be any solution for I that doesn't contain g_1, then by *Exchange Lemma* there is another solution OS' that contains g_1 and $|OS'| \leq |OS|$.

- We can write, $GS(I) = \{g_1\} \cup GS(I')$, given the design of the algorithm.
- Let $OS' = \{g_1\} \cup SomeSolution(I')$. OS' is a valid solution because g_1 covers all the customers in S and the rest of the customers(I') are covered by $SomeSolution(I')$. Note, $I' \cup S = I$.
- From induction hypothesis, we know $|GS(I')| \leq |SomeSolution(I')|$.
- From *Exchange Lemma*, we know $|OS'| \leq |OS|$.
- This implies, $|GS(I)| = 1 + |GS(I')| \leq 1 + |SomeSolution(I')| = |OS'| \leq |OS|$.

This completes the proof of the inductive step of the argument.

Having proved the correctness, we now need to give an efficient implementation of the greedy strategy and give time analysis.

(a) (5 points) Give an efficient algorithm implementing the above strategy, and give a time analysis for your algorithm.

Solution:

Algorithm 2 Placement of cell phone towers on a highway

```
1: function CellPhoneTowers(D[1, 2, ..., n], Δ)
2:     list = [ ]
3:     i = 1
4:     x = D[1] + Δ
5:     Append x to list
6:
7:     while (i ≤ n):
8:         while (D[i] ≤ x + Δ):
9:             i++
10:        if (i ≤ n):
11:            x = D[i] + Δ
12:            Append x to list
13:
14:    return list
15: end function
```

Running time analysis: i goes from 1 to n, so the algorithm precisely iterates once over the list D, making a single pass. Appending to the list takes $O(1)$ time and all other arithmetic and logical operations also take $O(1)$ time. Hence, the time is proportional to the size of the list. Therefore, the algorithm runs in $O(n)$ time.

5. (25 points) You are a conference organiser and you are asked to organise a conference for a company. The capacity of the conference room is limited and hence you would want to minimise the number of people invited to the conference. To make the conference useful for the entire company, you need to make sure that if an employee is not invited, then every employee who is an immediate subordinate of this employee gets the invitation (*if an employee is invited, then you may or may not invite a subordinate*). The company has a typical hierarchical tree structure, where every employee except the CEO has exactly one immediate boss.

   Design an algorithm for this problem. You are given as input an integer array $B[1...n]$, where $B[i]$ is the immediate boss of the i^{th} employee of the company. The CEO is employee number 1 and $B[1] = 1$. The output of your algorithm is a subset $S \subseteq \{1, ..., n\}$ of invited employees. Give running time analysis and proof of correctness.

Solution: The MinInvites function defined below returns the minimal set of people to be invited. It relies on a helper function BfsOrder which returns the order in which vertices are visited by BFS.

Running time analysis: Observe the following

- Lines 2, 9, 10 correspond to initialization of variable and take $O(1)$ time in total.
- Lines 3, 7 allocate arrays of size n and take $O(n)$ time.
- The for loop in lines 4-6 runs $n-1$ times and since appending to a linked list in line 5 is constant time, this loop takes $O(n)$ time.
- BfsOrder runs in $O(V+E)$ time in general since we can run BFS as usual and keep appending vertices to an array as they are discovered and return the array in the end. Therefore, line 8 runs in $O(V+E) = O(n)$ time since L represents a tree.
- The while loop in lines 12-22 runs $n-1$ times since i is decremented each time. Therefore, the lines 14-20 execute atmost $n-1$ times.
  - Since lines 14, 15 and 16 are individually $O(1)$ they take $O(n)$ time in total.
  - The for loop in lines 17-19 takes $O(\text{NumChildren}(x))$ time for each x. Note that after this loop, $P[c]$ = **false** for all i such that $B[c] = x$ since $L[x]$ contains precisely these c. Therefore, the loop 17-19 is always executed for different values of x on different iterations of the outer while loop. Thus, the time taken by this loop is at most $\sum_x O(\text{NumChildren}(x)) = O(n)$.

Since the total time taken by each line is atmost $O(n)$ we conclude that the overall function is $O(n)$ as well.

Algorithm 3 Finding minimum number of conference rooms required

```
1:  function MinInvites(B)
2:      n = |B|
3:      L = empty array of n linked lists
4:      for i = 2 to n do
5:          L[B[i]].append(i)
6:      end for
7:      P = array of size n with all values equal to true
8:      D = BfsOrder(L, 1)        ▷ Order in which vertices are discovered by BFS starting from 1
9:      R = empty linked list
10:     i = n
11:
12:     while i > 1 do
13:         if P[i] then
14:             x = B[D[i]]
15:             R.append(x)
16:             P[x] = false
17:             for c in L[x] do
18:                 P[c] = false
19:             end for
20:         end if
21:         i = i − 1
22:     end while
23:
24:     return R
25: end function
```

Proof of correctness: Note that the array P effectively stores which nodes are present in the tree. In each iteration of the while loop, the deepest node present in the tree is processed (since D contains vertices in ascending order of depth), its parent x (for $i > 1$, $x = B[i]$) is invited to the conference (x is appended to R) and then x and all its children are removed from the tree (P is set to **false**). i is then decremented and the same thing is repeated for the remaining nodes until just the CEO is left. We show the following exchange lemma

Lemma: For every solution OS there exists a solution OS' such that $|OS'| \leq |OS|$ with $x \in OS'$ and $c \notin OS'$ for all $c \in C_x$, where x is the parent of the deepest node in the tree and C_x are all the children of x.

Proof: Let x be the parent of the deepest node in the tree and C_x be the set of its children. Note each $c \in C_x$ is a leaf node. If some c had children then x cannot be the parent of the deepest node. Suppose $x \in OS$. Then, we can choose $OS' = OS \setminus C_x$ since once x is invited we need not invite its children i.e. OS' is a valid solution and clearly $|OS'| \leq OS$.

Now suppose $x \notin OS$, then all its children must be invited i.e. $C_x \subseteq OS$. This implies that we can set $OS' = (OS \cup \{x\}) \setminus C_x$ i.e. we can invite x and uninvite all its children. This still leaves us with a valid solution since for any node $y \neq x$ and $y \notin C_x$ we have not altered the membership of y or its children and therefore the desired property still holds. Whereas if $y = x$ or $y \in C_x$ we are covered since we have invited x. Furthermore, $|OS'| = |OS| + 1 - |C_x| \leq |OS|$ since $|C_x| \geq 1$. □

We can now show the main claim by induction.

Inductive hypothesis: For all trees with at most n nodes, the greedy solution is optimal

Base case: For $n = 1$, the greedy solution chooses to invite nobody which is clearly a valid solution. Further, it is optimal since 0 is the least number of people you could possibly invite.

Induction step: Suppose the IH holds for $n = k$ and consider a tree T with $k + 1$ vertices. Let OS be the optimal solution and $GS(T)$ be the greedy solution. Then, by the exchange lemma there exists a solution OS' such that the parent of the deepest node x (for $k > 1$ the deepest node always has a parent) is invited while all of its children C_x are not invited i.e. $x \in OS'$ and $C_x \cap OS' = \emptyset$. Further, since $|OS'| \leq |OS|$, OS' is optimal as well.

Now consider the tree T' which is obtained by removing x and all its children C_x from T. Consider the solution $OS'' = OS' \setminus \{x\}$. Clearly, OS'' only contains vertices that are in T'. Further, since for any $y \in T'$, y is also in T, either $y \in OS'$ or all its children in T $C_y \in OS'$. This in turn means that either $y \in OS''$ or all its children in T', $C''_y \in OS''$. Therefore, OS'' is a valid invitation for T'. Note that by the definition of the greedy strategy $GS = GS(T') \cup \{x\}$ where $GS(T')$ is greedy for T'. By the inductive hypothesis, $GS(T')$ is optimal for T' (since T' has atmost k nodes). This implies,

$$|OS'| = 1 + |OS''| \geq 1 + |GS(T')| = |GS(T)|$$

implying that $GS(T)$ is optimal for T. This completes the proof of the inductive step and the main claim.

6. (25 points) A town has n residents labelled $1, ..., n$. In the midst of a virus outbreak, the town authorities realise that hand sanitiser has become an essential commodity. They know that every resident in this town requires at least T integer units of hand sanitiser. However, at least $\lceil \frac{n}{2} \rceil$ residents do not have enough sanitiser. On the other hand, there may be residents who may have at least T units. With very few new supplies coming in for the next few weeks they want to implement some kind of sharing strategy. At the same time, they do not want too many people to get in close proximity to implement sharing. So, they come up with the following idea:

   Try to *pair up* residents (based on the amount of sanitiser they possess) such that:

   1. A resident is either unpaired or paired with exactly one other resident.
   2. Residents in a pair together should possess at least $2T$ units of sanitiser.
   3. The number of unpaired residents with less than T units of sanitzer is minimised.

   Once such a pairing is obtained, the unpaired residents with less than T units of sanitiser can be managed separately. The town authorities have conducted a survey and they know the amount of sanitiser every

resident possesses. You are asked to design an algorithm for this problem. You are given as input integer n, integer T, and integer array $P[1...n]$ where $P[i]$ is the number of units of sanitiser that resident i possesses. You may assume that $0 \leq P[1] \leq P[2] \leq ... \leq P[n]$. Your algorithm should output a pairing as a list of tuples $(i_1, j_1), (i_2, j_2), ..., (i_k, j_k)$ of maximum size such that (i) For all $t = 1, ..., k$, $P[i_t] + P[j_t] \geq 2T$ and (ii) $i_1, ..., i_k, j_1, ..., j_k$ are distinct. Give proof of correctness of your algorithm and discuss running time.

Solution:
The greedy strategy is to choose the unpaired resident with the highest amount of sanitiser, and match them with the unpaired resident with the lowest amount (< T) who satisfies sum ≥ 2T. More precisely:

Algorithm 4 Sanitiser Matching

```
1: function MatchSanitiser(P)
2:     S ← emptylist
3:     high ← n
4:     low ← 1
5:     while low < high and P[low] < T do
6:         if P[low] + P[high] ≥ 2T then
7:             Add (high, low) to S
8:             high ← high − 1
9:         end if
10:        low ← low + 1
11:    end while
12:    return S
```

Running Time: The loop increments *low* each time it runs and is conditioned on $low < high \leq n$, while the contents of the loop are constant time. Therefore the running time is $\mathcal{O}(n)$.

Proof of Correctness: The algorithm iterates through the list in ascending order until it finds a resident who can be paired with the unpaired resident with the highest sanitiser amount. It then pairs the two residents and runs again on the remaining set of residents (For efficiency reasons it does not retry for lower residents that have been passed over, since if they could not be paired with the highest resident, they cannot be paired with any resident). Therefore, the algorithm implements the above greedy strategy, and it suffices to prove correctness of the strategy. Let us define the cost of a solution as the number of unpaired residents with less that T amount sanitiser.

Lemma: If the first pairing made by the greedy solution GS is (i, j) then for any arbitrary solution OS, there exists a solution OS' such that OS' contains (i, j) and cost(OS') ≤ cost(OS).

Proof: If OS contains (i, j), OS' = OS. Otherwise, we have the following cases for OS:

1. Neither i nor j paired in OS: OS' includes (i, j) (which are unpaired in OS) as well as all OS pairings, therefore it is a valid solution, and since it has all OS pairings, cost(OS') ≤ cost(OS).
2. j unpaired but i paired with k in OS: OS' includes (i, j) and all OS pairings except the i, k pairing. OS' is valid since i, j are paired only once. $P[j] \leq P[k]$, since greedy chooses the lowest that can be matched with i. This means that if $P[k] < T$ then $P[j] < T$. Therefore if unpairing k increases cost, pairing i decreases it, so cost(OS') ≤ cost(OS).
3. i unpaired but j paired with k in OS: OS' includes (i, j) and all OS pairings except the j, k pairing. OS is valid since i, j are paired only once. Greedy chooses the lowest P-value resident that can be matched to i, so $P[j] \leq P[k]$. Since j and k are paired in OS, $2T \leq P[j] + P[k] \leq 2P[k] \rightarrow T \leq P[k]$. Therefore unpairing k does not increase cost, so cost(OS') ≤ cost(OS).
4. i paired with k and j paired with l in OS: OS' includes (i, j) and (k, l), as well as all OS pairings except the i, k and j, l pairings. Greedy chooses the lowest P-value resident that can be matched to i, so $P[j] \leq P[k]$. Since j and l are paired in OS, $2T \leq P[j] + P[l] \leq P[k] + P[l]$. Therefore (k, l) is a valid pairing. OS is therefore valid since i, j, l, k are paired only once. All i, j, k, l are paired, therefore there is no increase in cost, so cost(OS') ≤ cost(OS).

We note that if GS does not contain any pairing, then the highest P-valued resident cannot be paired with any other resident, which means that even the two highest P-valued residents cannot be paired. Therefore all valid solutions are empty and equivalent.

Claim: The greedy solution GS is valid and optimal.

Base Case: If $n = 0$ or $n = 1$ GS is empty and trivially correct.
Induction Hypothesis: Assume GS gives an optimal valid solution for $n \leq k$.
Indiction Step: If $n = k + 1$, suppose there exists some valid optimal solution OS(P). If GS(P) is empty, all solutions are empty and equivalent. Otherwise, by the above lemma there exists valid OS'(P) that includes the first greedy choice (i, j) and cost(OS'(P)) $\leq$ cost(OS(P)). Define P' as P with i and j removed. OS'(P) may therefore be written as $(i, j)+$ some solution OS'(P') for P', while GS(P) may be written as $(i, j)+$ GS(P') where GS(P') is the greedy solution for P'. The size of P' is $n - 2 = k + 1 - 2 = k - 1 \leq k$. Therefore by induction hypothesis GS(P') is valid and optimal, i.e. cost(GS(P')) $\leq$ cost(OS'(P')). cost(GS(P))= cost(GS(P')) $\leq$ cost(OS'(P')) = cost(OS'(P)) $\leq$ cost(OS(P)). In addition, (i, j) is only added if it is a valid pairing, therefore, GS(P) is a valid optimal solution.