Statement of Integrity: I, Derek Zhu, state that I completed this assignment with integrity and by myself.

Due: Jul 8, 2024

Problem 1:

I have two solutions:

Solution 1:

Function PostOrder(root)

```
Line 1:  If root is not empty then
Line 2:           PostOrder(root.left)
Line 3:           PostOrder(root.right)
Line 4:           Visit root
```

The runtime for each line is:
Line 1: run one time for each node, total n times;
Line 2: run one time for each node, total n times;
Line 3: run one time for each node, total n times;
Line 4: run one time for each node, total n times;
The total number of run time is: T(n) = n + n+ n+ n = 4n

Since $n < 4n < (4 + \delta)n,\ for\ \delta > 0\ and\ all\ n > 1,$
$then$: $T(n) = \Omega(n),\ and\ T(n) = O((4 + \delta)n) = O(n)$
Hence: $T(n) = \Theta(n)$ Proved!

Solution 2:

There are 2 recursive calls PostOrder(root.left) and PostOrder(root.right)
The size of each subproblem would be $1/2$ because of the nature of a binary search tree
The amount of work done at the end of each call is a constant
Using Master's theorem, with $a = 2,\ b = 2,\ f(n) = 1\ldots$ the recurrence relationship is:

$$T(n) = 2\,T(n/2) + O(1)$$

This falls in Case 1 of Master theorem,

so $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$ Proved!

Problem 2:

An AVL binary search tree (AVL BST) is a data structure that has the following properties:

1. Node Structure: Each node in a BST contains a key (or value) and pointers to its left and right children (which may be null).
2. Binary Tree: A BST is a binary tree, meaning each node has at most two children.
3. BST Property: For each node:
    (1) The key in the left child (and all nodes in the left subtree) is less than the key in the node.
    (2) The key in the right child (and all nodes in the right subtree) is greater than the key in the node.
4. In-order Traversal: An in-order traversal of a BST visits the nodes in ascending order of their keys. This means that if you perform an in-order traversal of a BST, you will retrieve the keys in sorted order.
5. Unique Keys: Typically, BSTs do not contain duplicate keys, meaning each key is unique within the tree.
6. For an AVL tree, the difference of the Height of Left Subtree and Height of Right Subtree of every node must be -1, 0, or 1.

Pseudocode:
function EXISTS_IN_RANGE(root, a, b)

```
#1    if root is empty then
#2        return false                              // no such key found
#3    if root.key < a then
#4        return EXISTS_IN_RANGE(root.right, a, b)  //not in left subtree, search right
#5    if root.key > b then
#6        return EXISTS_IN_RANGE(root.left, a, b)   //not in right subtree, search left
#7    return true    // a<=key<=b                   //if not left nor right nor empty,
                                                      then must be current key in
                                                      range
```

Runtime:
Given the fact that height of the tree is log(n),

Line #1:	log(n) in the worst case
Line #2	1
Line #3:	log(n) in the worst case
Line #4	log(n) in the worst case
Line #5	log(n) in the worst case
Line #6	log(n) in the worst case
Line #7	1

$Worst\ case\ runtime$: $T(n) = 5log(n) + 2$
$Hence$: $T(n) = O(log(n))$

Note: The best case runtime is constant runtime $T(n) = 1$, which is different from the worst runtime, so the tightest bound is $O(log(n))$ instead of $\Theta(log(n))$.

Correctness:

- Since all the keys in the left subtree are less than root.key, and all the keys in the right subtree are greater than the root.key according to BST's property…
- If the root.key is empty, just return false, meaning no such key exists.
- Therefore, if the root.key is less than the lower bound of the given range [a, b], it means the root.key < lower bound value a, then we just need to check the current right subtree and skip the current left subtree.
- Similarly, if the root.key is greater than the upper bound of the given range [a, b], it means the root.key > upper bound value b, then we just need to check the current left subtree and skip the current right subtree.
- If the root.key is in the given range [a, b], just return true, which means there exists such a key in the given range.

Problem 3:

My solution:

(a) Example: There are n = 4 containers of weight [3,3,1,1] units respectively. Standing on the deck is a set of trucks, each of which can hold K=4 units of weight.
By using the greedy algorithm, the 1st truck loads 3 unit weight, The 2nd truck loads 3+1=4 unit weight, The 3rd truck loads 1 unit weight, total needs 3 trucks.
By using an optimal way, the 1st truck loads 3+1=4 unit weight, the 2nd truck loads 3+1=4 unit weight, and only needs 2 trucks, instead of 3 trucks.
The lower bound or the minimum number of trucks can be calculated by: (total weight)/capacity
min_x= $(\sum_{i=1}^{i=n} w_i)/K = (3+3+1+1)/4 = 8/4 = 2$
This example shows this greedy algorithm does not use the minimum number of trucks.

(b) The lowest bound or the minimum number of trucks can be calculated by (total weight)/capacity:
$N_{min} = W/K.$
Consider the worst case: the weight is in such a sequence (w_1,w_2,w_3,...,w_{n-1},w_n) that
- truck 1 cannot load w_1 with w_2 together since $w_1+w_2 > K$ (only load w_1);
- truck $_2$ cannot load w_2 with w_3 together (only load w_2) since $w_2+w_3 > K$; …;
- truck i cannot load w_i with w_{i+1} together $w_i+w_{i+1} > K$ (only load w_i);

In another word, each truck can only load one container. Hence the number of trucks used by this greedy algorithm is $N_{greedy} = n$.

Let's add these inequalities together:
$(w1 + w2) + (w2 + w3) + \dots + (w_{n-1} + w_n) > (n-1)K$
Or: $(w_1+w_2+ ..+ w_n)+ (w_1+w_2+ ..+ w_n) -w_1-w_n > (n-1)K$
Let $w = w_1+w_2+ ..+ w_n$ which is the total weight of all containers.
Then we have:
$2W - (w_1+w_n) > (n-1)K$
Since $(w_1+w_n) > 0$
Hence: $2W > 2W - (w_1+w_n) > (n-1)K$
Therefore: $2W >(n-1)K$
Or : $(n-1) < 2W/K$
Or: $n < 1 + 2W/K$
Or: $N_{greedy} < 1 + 2N_{min}$

This shows that the number of trucks used by this greedy algorithm N_{greedy} is upper bounded or within a factor of two of the minimum possible number N_{min} for any set of weights and any value of K.

Problem 4:

My solution:

Define **DP(i)** as the minimum cost to ship the supplies from week 1 to week i. Consider the following cases for week i:
If Company A is used for week i, then the cost for this week is: $r * s_i$, and the total cost is the cost up to week (i-1) plus the cost of using Company A for week i: $DP(i) = DP(i - 1) + r * s_i$
If Company B is used for weeks i-3 to i, note: this case is only valid if $i \geq 4$, because it doesn't make sense to use Company B for less than four contiguous weeks when the cost for these 4 weeks is const c.
The total cost is the cost up to week (i-4) plus the cost of using Company B for the block of 4 weeks:
$DP(i) = DP(i - 4) + c$
Let's combine these cases to form the Bellman equation:
Base Cases: For i=0 (no weeks):

- DP(0) = 0;
- For $i = 1$: Company B cannot be chosen because $i < 4$. Hence only Company A can be used:
  $$DP(1) = r * s_1 = \sum_{j=1}^{1} r * s_j$$
- For $i = 2$: Company B cannot be chosen because $i < 4$. Hence only Company A can be used:
  $$DP(2) = DP(1) + r * s_2 = r * s_1 + r * s_2 = \sum_{j=1}^{2} r * s_j$$
- For $i = 3$: Company B cannot be chosen because $i < 4$. Hence only Company A can be used:
  $$DP(3) = DP(2) + r * s_3 = r * s_1 + r * s_2 + r * s_3 = \sum_{j=1}^{3} r * s_j$$

Other cases (For $i \geq 4$):
Let cost A= $DP(i - 1) + r * s_i$, represent the cost of using Company A for week i added to the minimum cost up to week i-1;
Let cost B = $DP(i - 4) + c$, represent the cost of using Company B for the block of weeks (i-3) to i, added to the minimum cost up to week (i-4);
Then by using dynamic programming:
$DP(i) = min\{using\ cost_A\ ,\ using\ Cost_B\}$
or: $DP(i) = min\{DP(i - 1) + r * s_i\ ,\ DP(i - 4) + c\}$

Summary of the Bellman equation we derived:
$DP(0) = 0,\ if\ i = 0$;
$DP(i) = \sum_{j=1}^{i} r * s_j$, if $1 \leq i \leq 3$;
$DP(i) = min\{DP(i - 1) + r * s_i\ ,\ DP(i - 4) + c\}$, if $4 \leq i \leq n$;

Problem 5:

My solution:

(a) Recursive Brute-Force Algorithm

Idea: to explore all possible permutations paths to determine the shortest one.

Pseudocode:
Input: list of lighthouse names L; starting lighthouse s guaranteed not to be in L; dictionary of travel times TRAVEL_TIME[Li , Lj] for every pair of lighthouses in L;
Output: ordered list of lighthouses corresponding to the fastest (shortest) possible path starting at s, and travel time;
function FASTEST_PATH(s, L, TRAVEL_TIME):

```
        // Base case: If lighthouse list L is empty, return an empty path and zero travel time
#1      if L is empty:
#2          return empty, 0
#3      fastest_path ← None
#4      fastest_time ←infinity
        // Try each lighthouse as the next step
#6      for next_light in L:
            // Calculate remaining lighthouses
#7          remaining_lights ← L.copy()
#8          remaining_lights.remove(next_light)
            // Recursive call to find the fastest path from next_light to remaining_lights
#9          sub_path, sub_time = FASTEST_PATH(next_light, remaining_lights, TRAVEL_TIME)
            // Calculate the total time for this path
#10         total_time ← TRAVEL_TIME[s][next_light] + sub_time
            // Check if this path is the fastest
#11         if total_time < fastest_time:
#12             fastest_time ← total_time
#13             fastest_path ← [next_light] + sub_path
#14     return fastest_path, fastest_time
```

Running Time Analysis:

The running time of this algorithm is factorial in nature because it explores all permutations of the lighthouses to find the optimal path. Specifically, for n lighthouses, the time complexity is O(n!), where n is the number of lighthouses in the list L.

- Recursion Depth: The recursion depth is equal to the number of lighthouses, n, as each recursive call reduces the list of remaining lighthouses by one.
- Number of Recursive Calls: There are n! permutations of the lighthouses, and the algorithm evaluates each one.

Correctness Analysis:

This recursive brute-force approach ensures that all possible paths are evaluated to find the shortest path, which guarantees the optimal solution for the given problem.

(b) Kickoff Function with Phantom Lighthouse algorithm

Idea: modify the data structure by adding a phantom lighthouse p to the lighthouse list with a travel time of 0 to the TRAVEL_TIME dictionary, then call the above Recursive Brute-Force Algorithm by starting the phantom lighthouse P to find the shortest path. Then get the final answer by removing the phantom lighthouse p from the shortest path found in Recursive Brute-Force Algorithm.

Pseudocode:
Input: list of lighthouse names L; dictionary of travel times TRAVEL_TIME[Li , Lj] for every pair of lighthouses in L;
Output: ordered list of lighthouses corresponding to the fastest (shortest) possible path, and travel time;

```
function KICKOFF(L, TRAVEL_TIME):
// Add the phantom lighthouse
#1    L_with_phantom ← L.copy()
#2    L_with_phantom.append('p')
// Add travel times from the phantom lighthouse to all other lighthouses
#2    for light in L:
#3        TRAVEL_TIME['p'][light] ← 0
#4        TRAVEL_TIME[light]['p'] ← 0
// Call FASTEST_PATH with the phantom lighthouse as the start
#5    best_path, best_time ← FASTEST_PATH('p', L, TRAVEL_TIME)
// Remove the phantom lighthouse from the path
#6    best_path.remove('p')
#7    return best_path, best_time
```

Running Time Analysis:
Line#1: 1
Line#2: 1
Line#3: n
Line#4: n
Line#5: O(n!)
Line#6: 1
Line#7: 1
$T(n) = 4 + 2n + O(n!) = O(n!)$

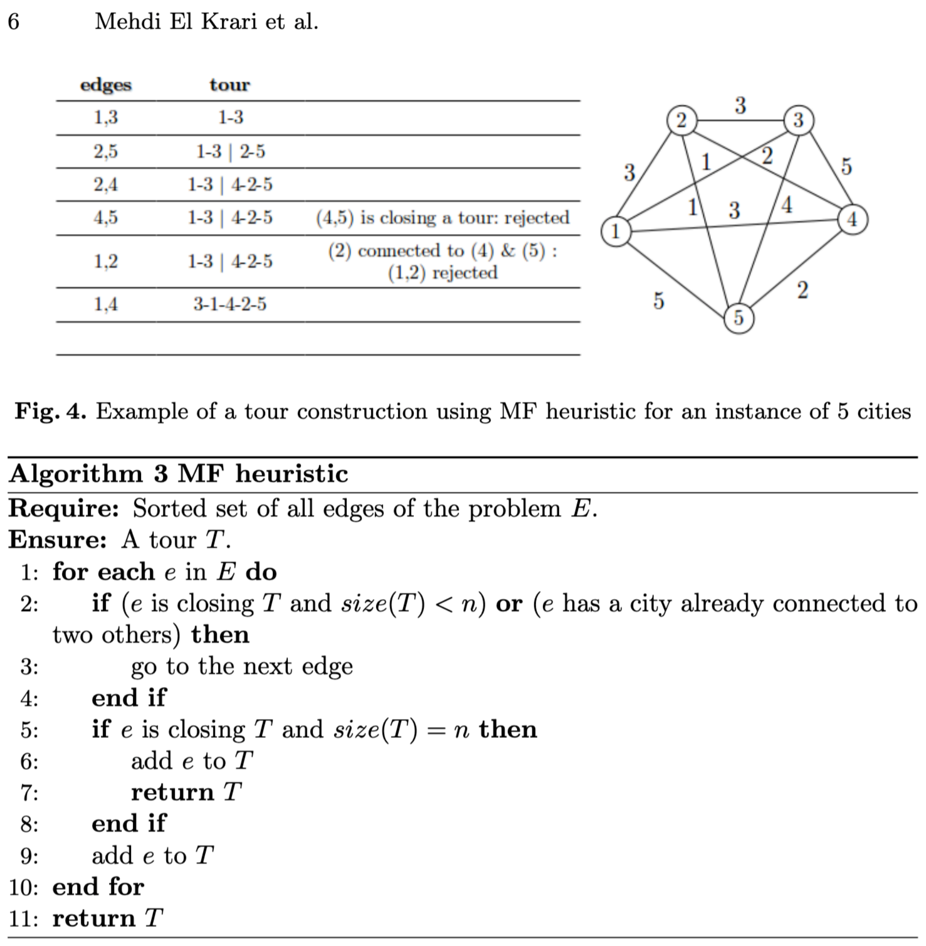
Correctness Analysis: Add a phantom lighthouse p that is connected to every other lighthouse with a travel time of 0, this wouldn't change the travel time for any paths. Call FASTEST_PATH starting from the phantom lighthouse, and then remove the phantom lighthouse from the resulting path to get the best starting lighthouse automatically. Therefore, the added phantom lighthouse will not change any paths. Hence the shortest path found is the same as the shortest path found in the Brute-Force Algorithm, which's correctness has been analyzed.

(c) Multi-Fragment Algorithm

Idea:

since the Multi-Fragment (MF) solves the problem for a tour, which includes going back to start, but that we are interested in a path that does not return to start, therefore we cannot use that MF algorithm directly to solve our problem. If we can modify the MF algorithm by not including going back to the start part then it will solve our problem.

The MF algorithm is shown in the following picture (copied from the paper mentioned in PA3 here, named "An Empirical Study of the Multi-Fragment Tour Construction Algorithm.pdf "):



6 Mehdi El Krari et al.

edges	tour	
1,3	1-3	
2,5	1-3 | 2-5	
2,4	1-3 | 4-2-5	
4,5	1-3 | 4-2-5	(4,5) is closing a tour: rejected
1,2	1-3 | 4-2-5	(2) connected to (4) & (5) : (1,2) rejected
1,4	3-1-4-2-5	

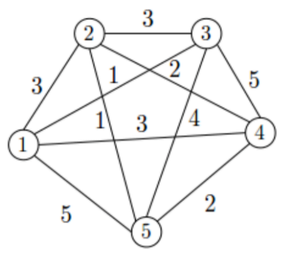


Fig. 4. Example of a tour construction using MF heuristic for an instance of 5 cities

Algorithm 3 MF heuristic

```
Require: Sorted set of all edges of the problem $E$.
Ensure: A tour $T$.
1: for each $e$ in $E$ do
2:     if ($e$ is closing $T$ and $size(T) < n$) or ($e$ has a city already connected to two others) then
3:         go to the next edge
4:     end if
5:     if $e$ is closing $T$ and $size(T) = n$ then
6:         add $e$ to $T$
7:         return $T$
8:     end if
9:     add $e$ to $T$
10: end for
11: return $T$
```

If we comment out line #6, then the returned travel path T will not include going back to the start segment. Hence the modified algorithm will be:

Pseudocode 1:

Input: list of lighthouse names L; dictionary of travel times TRAVEL_TIME[Li , Lj] for every pair of lighthouses in L;
Output: ordered list of lighthouses corresponding to the fastest (shortest) possible path, and travel time

```
function MF_complete(L, TRAVEL_TIME ):
Line 1:  Edges ← sortedEdges(L, TRAVEL_TIME)
Line 2:  Shortest_edges ← MF_heuristic(edges)
Line 3:  best_path ← edges_to_path(shortest_edges)
Line 4: best_time ← calculate_total_travel_time(best_path, TRAVEL_TIME)
Line 5: return best_path, best_time
```

Runtime Analysis:

Line 1: sortedEdges(L, TRAVEL_TIME): As analyzed below, this has a runtime of $O(n^2 logn)$

Line 2: MF_heuristic(edges): As analyzed below, this has a runtime of $O(n^2)$.

Line 3: edges_to_path(shortest_edges): As analyzed below, this has a runtime of $O(n^2)$.
Line 4: calculate_total_travel_time(best_path, TRAVEL_TIME): As analyzed below, this has a runtime of O(n).
Line 5: O(1)

Combining these: the overall runtime of MF_complete is $O(n^2 logn)$

Pseudocode 2:

Input: list of lighthouse names L; dictionary of travel times TRAVEL_TIME[Li , Lj] for every pair of lighthouses in L;
Output: sorted edge list. Each edge is a pair of lighthouse' names. The edge list sorted by using travel time between the two lighthouses;
function sortedEdges(L, TRAVEL_TIME):

```
Line 1:  E ← empty
Line 2:  for i = 1 to len(L):
Line 3:          for j =i+1 to len(L):
Line 4:                  Current_edge.x  ← L[i]
Line 5:                  Current_edge.y  ← L[j]
Line 6:                  Current_edge.time =TRAVEL_TIME[Current_edge.x][Current_edge.y]
                         //use INSERTION-SORT from CLRS Chapter 2:
Line 7:                  INSERTION-SORT current_edge to E  and sort by time
Line 8: Return E
```

Runtime Analysis

- Edge Generation: Generating edges takes $O(n^2)$ time because there are n(n−1)/2≈$O(n^2)$ pairs.
- INSERTION-SORT: Each insertion operation for insertion sort in the worst case can take $O(n^2)$ time for sorting all pairs.Since there are $O(n^2)$ edges, the insertion sort would take $O(n^2 * n^2) == O(n^4)$ in the worst case. However, if we assume sorting all the edges initially, the complexity would be $O(n^2 logn)$ using efficient sorting algorithms like merge sort or quicksort.

Combining these:

- The overall runtime of sortedEdges is $O(n^2 logn)$ if we use an efficient sorting algorithm instead of insertion sort.

Pseudocode 3

```
Input: a sorted list of edges;
Output: the fastest (shortest) possible edges
function MF_heuristic(E):
Line 1: T ← empty
//ensure the travel path is a valid path:
Line 2:  for each edge in Edges do
Line 3:           if is_edge_closing_path (T, edge) and size(T) < n
Line 4:                   continue
Line 5:           if (edge has a node already connected to two others)
Line 6:                   continue
Line 7:           if  is_edge_closing_path (T, edge) and size(T) = n
//I intent to comment out this line from the original algorithm in the paper:
                          // add edge to T
Line 8:                   return T
Line 9:           Add edge to T
Line 10:return T
```

Runtime Analysis:

Line 1: Initialization: O(1)

Line 2~9: Loop through edges: The outer loop runs $O(n^2)$ times because there are $O(n^2)$ edges.

Conditions inside loop: Each condition and operation inside the loop can be considered O(1).

Line 10: O(1)

Combining these: the overall runtime of MF_heuristic is $O(n^2)$.

Pseudocode 4

Input: a list of edges;
Output: a valid path of the edges
function edges_to_path(edges):

```
// Step 1: Count occurrences of each node
Line 1:  node_count ← {}
Line 2:  for e in edges:
Line 3:          if e.x in node_count:
Line 4:                  node_count[e.x] ← node_count[e.x] + 1
Line 5:          else:
Line 6:                  node_count[e.x] ← 1
Line 7:          if e.y in node_count:
Line 8:                  node_count[e.y]  ← node_count[e.y] + 1
Line 9:          else:
Line 10:                 node_count[e.y] ← 1

// Step 2: Find the starting node (the node with only one occurrence)
Line 11:start_node = None
Line 12:for node in node_count:
Line 13:         if node_count[node] == 1:
Line 14:                 start_node ← node
Line 15:                 break

// Step 3: Construct the path
Line 16:path ←  []
Line 17:current_node ← start_node
Line 18:visited_edges ← empty set
Line 19:while len(visited_edges) < len(edges):
Line 20:         path.append(current_node)
Line 21:         for edge in edges:
Line 22:                 if edge not in visited_edges:
Line 23:                         x, y ← edge.x, edge.y
Line 24:                 if x == current_node:
Line 25:                         visited_edges.add(edge)
Line 26:                         current_node ← y
Line 27:                          break
Line 28:                 else if y == current_node:
Line 29:                         visited_edges.add(edge)
Line 30:                         current_node ← x
Line 31:                          break
Line 32:                 path.append(current_node)  // Append the last node to complete the path
Line 33: return path
```

Runtime Analysis:

Line 1~10: Count Occurrences: $O(n^2)$ because we iterate through all edges at once.
Line 11~15: Find Starting Node: O(n) because we iterate through all unique nodes.
Line 16~33: Construct Path:The outer while loop runs O(n) times because we have to visit each node once.The inner loop (searching for the next edge) runs $O(n^2)$ times in the worst case if we check all edges each time.

Combining these: the overall runtime of edges_to_path is $O(n^2)$

Pseudocode 5

Input: a valid path of the edges; dictionary of travel times TRAVEL_TIME[Li , Lj] for every pair of lighthouses in L;
Output: the travel time of the path
function calculate_total_travel_time(path, TRAVEL_TIME):

```
// Initialize total travel time to zero
Line 1:  total_time ← 0

// Iterate through the path, considering each consecutive pair of nodes
Line 2:  for i in range(len(path) - 1):
Line 3:          u ← path[i]        // Current node in the path
Line 4:          v ← path[i + 1]  // Next node in the path

// Add the travel time between the current node and the next node to the total time
Line 5:          total_time ←  total_time + TRAVEL_TIME[u][v]

// Return the total travel time after iterating through the entire path
Line 6:  return total_time
```

Runtime Analysis:
Line 1: Initialization: O(1)
Line 2~4: Iteration through Path: The loop runs O(n) times because we iterate through each node in the path.
Line 5~6: O(1)
Combining these: the overall runtime of calculate_total_travel_time is O(n).

Pseudocode 6:

Input: T - list of tuples, where each tuple (u, v) represents an edge in the current path. edge - tuple (u, v), representing the new edge to be added.
Output: Returns True if adding the edge forms a cycle, otherwise False.
Helper function to check if adding an edge will form a cycle.
Function is_edge_closing_path(T, edge):

```
// note:  make_set(x),  union(x, y) are the algorithm from CLRS book page 530
Line 1:         parent ← {}
# Create sets for the nodes in the current path
Line 2:         nodes ← empty set
Line 3:         for u, v in T:
Line 4:             nodes.add(u)
Line 5:             nodes.add(v)
Line 6:             if u not in parent:
Line 7:                 call make_set(u)
Line 8:             if v not in parent:
Line 9:                 call make_set(v)
Line 10:            call union(u, v)
# Check if the new edge forms a cycle:
Line 11:        u, v ←  edge
Line 12:        if u not in parent:
Line 13:            call make_set(u)
Line 14:        if v not in parent:
Line 15:            call make_set(v)
Line 16:        return call union(u, v)
```

Runtime Analysis:
Line 1~2: Initialization: O(1)
Line 3~10: Iteration through Path: The loop runs O(n) times because we iterate through each node in the path.
Line 11~16: O(1)
Combining these: the overall runtime of calculate_total_travel_time is O(n).