CS 6/L

Algorithms and Complexities

Final Project

Submitted by:

Canlas, Rey Vincent

Blase, Alexis

Nebran, Bern Homer

Talamillo, Dominic

Submitted to:

Prof. Daniel Ryan Quiño

May 21, 2021

1. Fair Property Division

In this problem, we implemented an algorithm that is somewhat similar to **Knapsack algorithm** where the goal is to look for the maximum value to a specific knapsack (in this case subset) while remaining within the weight constraint. The way the algorithm solves this problem is to get first the total sum of card values, then divide it into two sets (set 1 for Abe and set 2 for Bob), and then it is the job of the algorithm to identify which cards to take for both sets to divide the cards as evenly as possible.

The following is the process on how the algorithm solves the problem:

1.1. The algorithm starts by adding all the values of the cards, then divide by 2 since we have two sets.

2 + 1 + 3 + 1 + 5 + 2 + 3 + 4 = 21

21 / 2 = 10.5 or 10 (This is now the capacity to carry on each sets)

1.2. Start performing the tabulation process (dynamic programming). By default, set rows [0][i] to false and [i][0] to true.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	**0**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
1	**2**	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
2	**1**	**T**										
3	**3**	**T**										
4	**1**	**T**										
5	**5**	**T**										
6	**2**	**T**										
7	**3**	**T**										
8	**4**	**T**										

1.3. (1st Card) Since card 1 has a value of 2, **therefore the maximum card values to take is 2**.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	**0**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
1	**2**	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
2	**1**	**T**										
3	**3**	**T**										
4	**1**	**T**										
5	**5**	**T**										
6	**2**	**T**										
7	**3**	**T**										
8	**4**	**T**										

1.4. (2nd Card) Since card 1 has a value of 2 and card 2 has a value of 1, **therefore the maximum card values to take is 3**. So those values that are less than and equal to 3 are set to true, and those that are greater than 3 set to false.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	0	T	F	F	F	F	F	F	F	F	F	F
1	2	T	F	T	F	F	F	F	F	F	F	F
2	1	T	T	T	T	F	F	F	F	F	F	F
3	3	T										
4	1	T										
5	5	T										
6	2	T										
7	3	T										
8	4	T										

1.5. (3rd Card) Repeat the same process by adding the values of previous cards (Card 1 and Card 2) and current card (Card 3). The total sum of cards is 6 (2 + 1 + 3), **therefore the maximum card values to take is 6**. So those values that are less than and equal to 6 are set to true, and those that are greater than 6 set to false.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	0	T	F	F	F	F	F	F	F	F	F	F
1	2	T	F	T	F	F	F	F	F	F	F	F
2	1	T	T	T	T	F	F	F	F	F	F	F
3	3	T	T	T	T	T	T	T	F	F	F	F
4	1	T										
5	5	T										
6	2	T										
7	3	T										
8	4	T										

1.6. (4th Card) Repeat the same process by adding the values of previous cards (Card 1, Card 2, Card 3) and current card (Card 4). The total sum of cards is 7 (2 + 1 + 3 + 1), **therefore the maximum card values to take is 7**. So those values that are less than and equal to 7 are set to true, and those that are greater than 7 set to false.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	0	T	F	F	F	F	F	F	F	F	F	F
1	2	T	F	T	F	F	F	F	F	F	F	F
2	1	T	T	T	T	F	F	F	F	F	F	F
3	3	T	T	T	T	T	T	T	F	F	F	F
4	1	T	T	T	T	T	T	T	T	F	F	F
5	5	T										
6	2	T										
7	3	T										
8	4	T										

1.7. (5th Card) Repeat the same process by adding the values of previous cards (Card 1, Card 2, Card 3, Card 4) and current card (Card 5). The total sum of cards is 12 (2 + 1 + 3 + 1 + 5), **therefore the maximum card values to take is 12**. So those values that are less than and equal to 12 are set to true.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	0	T	F	F	F	F	F	F	F	F	F	F
1	2	T	F	T	F	F	F	F	F	F	F	F
2	1	T	T	T	T	F	F	F	F	F	F	F
3	3	T	T	T	T	T	T	T	F	F	F	F
4	1	T	T	T	T	T	T	T	T	F	F	F
5	5	T	T	T	T	T	T	T	T	T	T	T
6	2	T										
7	3	T										
8	4	T										

1.8. (6th – 8th Card) Set all the values to true since the maximum card values already exceeds the capacity.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	0	T	F	F	F	F	F	F	F	F	F	F
1	2	T	F	T	F	F	F	F	F	F	F	F
2	1	T	T	T	T	F	F	F	F	F	F	F
3	3	T	T	T	T	T	T	T	F	F	F	F
4	1	T	T	T	T	T	T	T	T	F	F	F
5	5	T	T	T	T	T	T	T	T	T	T	T
6	2	T	T	T	T	T	T	T	T	T	T	T
7	3	T	T	T	T	T	T	T	T	T	T	T
8	4	T	T	T	T	T	T	T	T	T	T	T

1.9. After the tabulation, the tracing process begins. Starting on the right most corner of our table (Row 8 Column 11). We set red as Bob's Cards, and Yellow to Abe's Cards.

Card	Value	0	1	2	3	4	5	6	7	8	9	10
0	0	T			F	F	F	F	F	F	F	F
1	2	T	F	T	F	F	F	F	F	F	F	F
2	1	T	T	T				F	F	F	F	F
3	3	T	T	T	T	T	T	T	F	F	F	F
4	1	T	T	T	T	T	T	T				F
5	5	T	T	T	T	T	T	T	T	F	F	F
6	2	T	T	T	T	T	T	T	T	T		T
7	3	T	T	T	T	T	T	T	T	T	T	T
8	4	T	T	T	T	T	T	T	T	T	T	T

2 steps back.

4 steps back.

6 steps back.

Abe's Baseball Cards = {C5, C3, C1}

Bob's Baseball Cards = {C8 C7, C6, C4, C2}

2. Bridge to Nowhere

In this problem, we implemented the **longest common subsequence** in which it will get the maximum sequence that can be obtained with the two arrays where in this case the first array contains the cities on the north side and the second array contains the cities on the south side.

The following is the process on how the algorithm solves the problem:

north[i] = {1, 2, 3, 4, 5, 6}

south[i] = {4, 5, 1, 3, 6, 2}

			1	2	3	4	5	6
		0	1	2	3	4	5	6
	0							
4	1							
5	2							
1	3							
3	4							
6	5							
2	6							

2.1. Start performing the tabulation process (dynamic programming). By default, set the values of column [i][0] and rows [0][i] to zero.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0						
5	2	0						
1	3	0						
3	4	0						
6	5	0						
2	6	0						

2.2. (Second Row / City 4) Starting with city number 4, check each cell if city 4 also exists on north[i]. If north[i] == south[i], then mark the cell with 1 + its diagonal element value, otherwise mark the max value between the top value and its previous value. Since city 4 matches with one of the cities in north[i], mark the cell with 1 + its diagonal element value. So, 0 + 1 = 1.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0				1		
5	2	0						
1	3	0						
3	4	0						
6	5	0						
2	6	0						

2.3. Those cities on the north side that are less than 4, set the values to zero and those that are greater and equal to 4 set to the maximum value, which is 1.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0						
1	3	0						
3	4	0						
6	5	0						
2	6	0						

2.4. (Third Row / City 5) Next is city number 5, check each cell if city 5 also exists on north[i]. If north[i] == south[i], then mark the cell with 1 + its diagonal element value, otherwise mark the max value between the top value and its previous value. Since city 5 matches with one of the cities in north[i], mark the cell with 1 + its diagonal element value. So, 1 + 1 = 2.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0					2	
1	3	0						
3	4	0						
6	5	0						
2	6	0						

2.5. Those cities on the north side that are less than city 5, mark the max value between the top value and its previous value. On the other hand, those cities that are greater or equal to city 5 set to the maximum value.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0						
3	4	0						
6	5	0						
2	6	0						

2.6. (Fourth Row / City 1) Next is city number 1, check each cell if city 1 also exists on north[i]. If north[i] == south[i], then mark the cell with 1 + its diagonal element value, otherwise mark the max value between the top value and its previous value. Since city 1 matches with one of the cities in north[i], mark the cell with 1 + its diagonal element value. So, 0 + 1 = 1.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1					
3	4	0						
6	5	0						
2	6	0						

2.7. Those cities on the north side that are less than city 1, mark the max value between the top value and its previous value. On the other hand, those cities that are greater or equal to city 1 set to the maximum value.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1	1	1	1	2	2
3	4	0						
6	5	0						
2	6	0						

2.8. (Fifth Row / City 3) Next is city number 3, check each cell if city 3 also exists on north[i]. If north[i] == south[i], then mark the cell with 1 + its diagonal element value, otherwise mark the max value between the top value and its previous value. Since city 3 matches with one of the cities in north[i], mark the cell with 1 + its diagonal element value. So, 1 + 1 = 2.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1	1	1	1	2	2
3	4	0			2			
6	5	0						
2	6	0						

2.9. Those cities on the north side that are less than city 3, mark the max value between the top value and its previous value. On the other hand, those cities that are greater or equal to city 3 set to the maximum value.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1	1	1	1	2	2
3	4	0	1	1	2	2	2	2
6	5	0						
2	6	0						

2.10. (Sixth Row / City 6) Next is city number 6, check each cell if city 6 also exists on north[i]. If north[i] == south[i], then mark the cell with 1 + its diagonal element value, otherwise mark the max value between the top value and its previous value. Since city 6 matches with one of the cities in north[i], mark the cell with 1 + its diagonal element value. So, 2 + 1 = 3.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1	1	1	1	2	2
3	4	0	1	1	2	2	2	2
6	5	0						3
2	6	0						

2.11. Those cities on the north side that are less than city 6, mark the max value between the top value and its previous value. On the other hand, those cities that are greater or equal to city 6 set to the maximum value.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1	1	1	1	2	2
3	4	0	1	1	2	2	2	2
6	5	0	1	1	2	2	2	3
2	6	0						

2.12. (Seventh Row / City 2) Next is city number 2, check each cell if city 2 also exists on north[i]. If north[i] == south[i], then mark the cell with 1 + its diagonal element value, otherwise mark the max value between the top value and its previous value. Since city 2 matches with one of the cities in north[i], mark the cell with 1 + its diagonal element value. So, 1 + 1 = 2.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1	1	1	1	2	2
3	4	0	1	1	2	2	2	2
6	5	0	1	1	2	2	2	3
2	6	0		2				

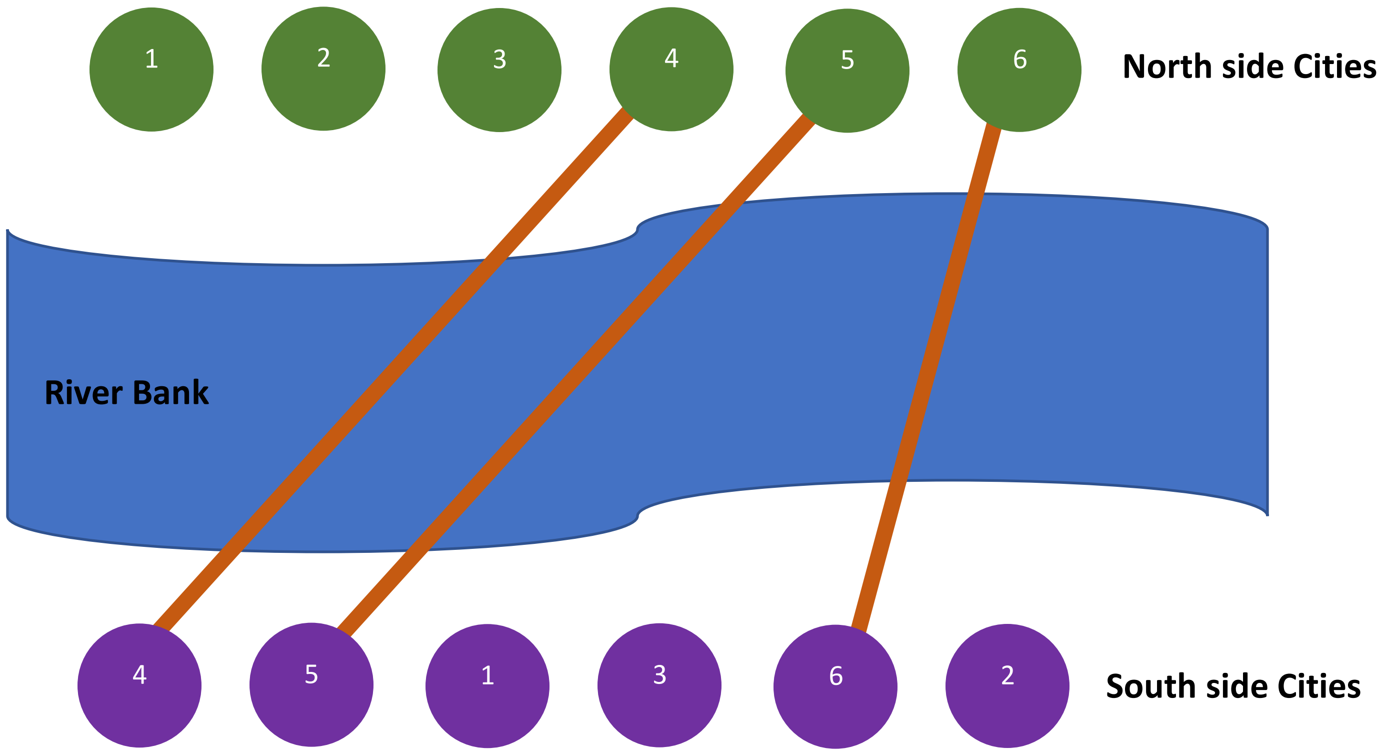
2.13. Those cities on the north side that are less than city 2, mark the max value between the top value and its previous value. On the other hand, those cities that are greater or equal to city 2 set to the maximum value.

	North		1	2	3	4	5	6
South		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	1	1
5	2	0	0	0	0	1	2	2
1	3	0	1	1	1	1	2	2
3	4	0	1	1	2	2	2	2
6	5	0	1	1	2	2	2	3
2	6	0	1	2	2	2	2	3

2.14. This is now the maximum number of bridges that can be build, and the cities in where to build them.

	North		1	2	3	4	5	6	
South		0	1	2	3	4	5	6	
	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	1	1	1	City 4
5	2	0	0	0	0	1	2	2	City 5
1	3	0	1	1	1	1	2	2	
3	4	0	1	1	2	2	2	2	
6	5	0	1	1	2	2	2	3	City 6
2	6	0	1	2	2	2	2	3	

Maximum Number of Bridges: 3



3. Travel Buddies

In this problem, we implemented the **Prim's Algorithm** which is a type of greedy algorithm where it finds the minimum spanning tree of a specific weighted and undirected graph. The way it solves this problem is that it will start to an arbitrary vertex where it locates the cheapest path from the other vertex. One of the reasons why this algorithm is good for solving this kind of problem it is because it was stated that the goal was to find the minimum travel cost and also visiting all the cities is required without going back to the starting point (the tour ends in the last city).

The following is the process on how the algorithm solves the problem:

3.1. Divide the number of cities into two sub-list (one for you and your friend)

n = 6 (total cities) with a total of 36 paths
A = 3 (Your list) with a total of 9 paths
B = 3 (Your friend's list) with a total of 9 paths

Note:

City 1, 2, 3 (Cities you will visit)
City 4, 5, 6 (Cities your friend will visit)

3.2. Get the paths from your city and your friend's city, to optimize travel cost

City	1	2	3	4	5	6
1	0	2	10	3	11	13
2	2	0	13	4	11	15
3	10	13	0	9	2	3
4	3	4	9	0	8	12
5	11	11	2	8	0	4
6	13	15	3	12	4	0

🟧 - The paths of the cities you will visit. 🟩 - The paths of the cities your friend will visit.

3.3. Store the paths on a two-dimensional array.

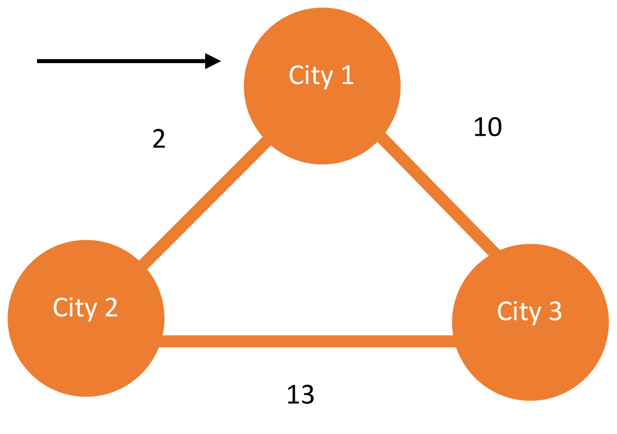
A[][]=

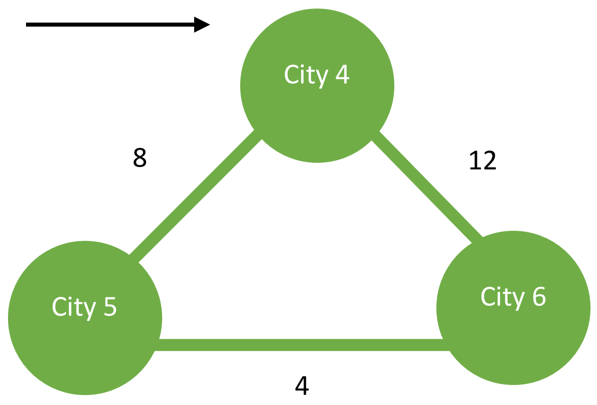
City	1	2	3
1	0	2	10
2	2	0	13
3	10	13	0

B[][]=

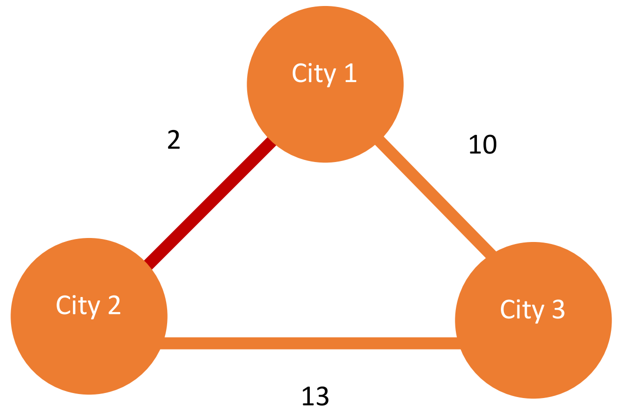
City	4	5	6
4	0	8	12
5	8	0	4
6	12	4	0

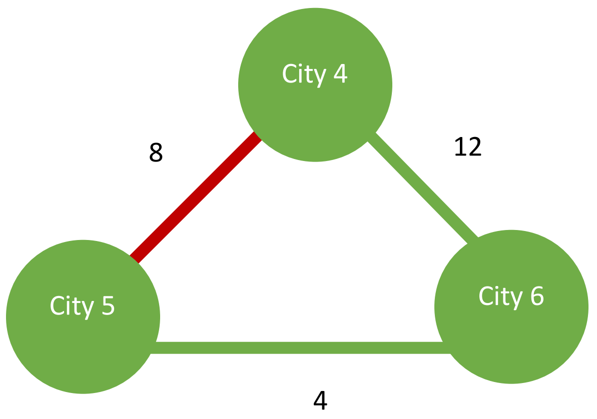
3.4. Apply Prim's algorithm on both arrays. For A, choose an arbitrary vertex, in this case choose City #1. On the other hand, for B, also choose an arbitrary vertex, which is City #4.



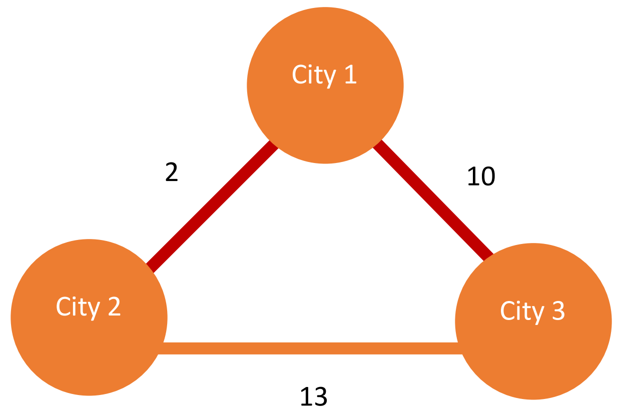


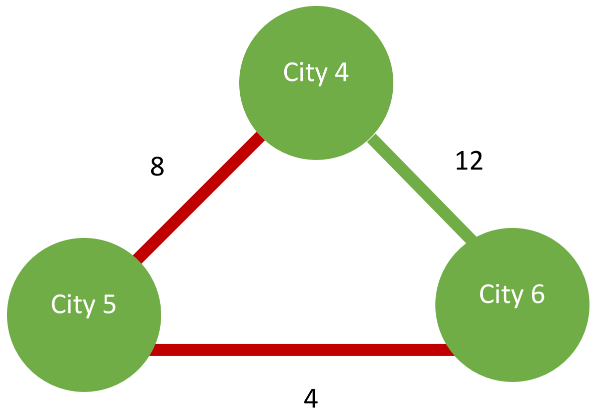
3.5. In City #1, it can visit 2 cities (City 2 and City 3). From City 1 to 2, it has a cost of 2, as for City 1 to 3, it has a cost of 10. So, in this case choose the ***least costly edge*** which is from **City 1 to 2.** Same goes with City 4, from City 4 to 5, it has a cost of 8, and for City 4 to 6 it has a cost of 12. Therefore, choose the path **from City 4 to 5**.



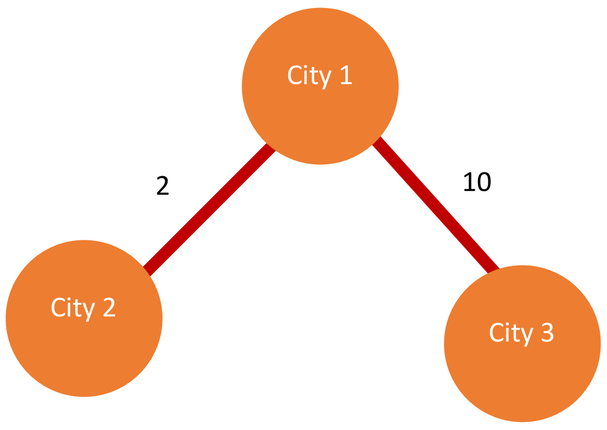


3.6. We are now in City #2, which has two paths (From City 2 to 3 and City 1 to 3). Since the least the least costly edge is 10, then we take the path from City 1 to 3. The same with City #5, which also has two paths and we take the path from City 5 to City 6 because the edge cost is 4.

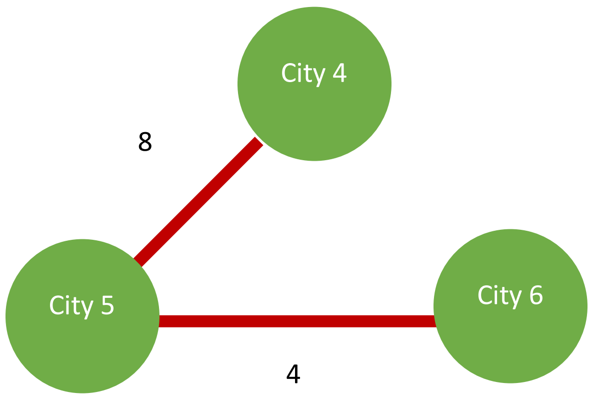




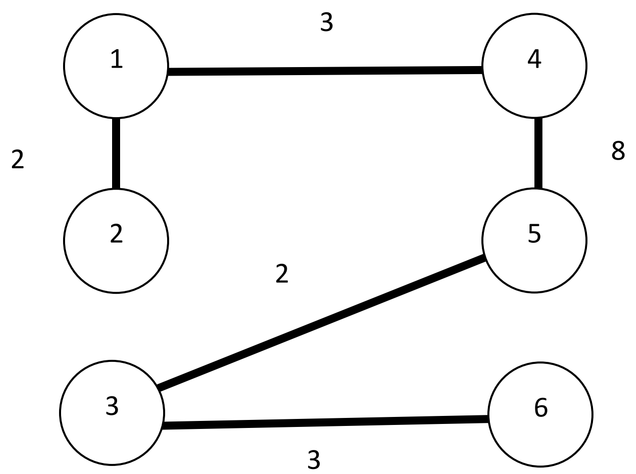
3.7. Since all the cities are visited, this is now the minimum spanning tree with the optimal path. In addition, we cannot go any further since going back to City #1 and City #4 will form a cycle.



Total Cost of your Path: 12



Total Cost of your Friend's Path: 12



Total Cost without splitting the list in to two sub-list: 18

4. Bus Stop Prize

In this problem, I implemented Kadane's algorithm in order to get the subarray with the largest sum. The goal of this algorithm is to get the maximum sum of contiguous subarrays (meaning a set of any indices that are adjacent to each other) to get the optimal solution.

The following is the process on how the algorithm solves the problem:

A = {4, -1, 5, -4, 2, 6, -8, -2, 9, -3} (size = 10)

4.1. The algorithm starts by setting the first element of array A[i] as the current value and the global value. So in this case, we initialize both variables to 4, since A[0] = 4.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4									
global	4									

4.2. Next is index 1, identify the maximum sub-array ending at this index. It is either ***sub-array [1]*** which is equal to -1 or [0, 1] which is equal to -1 + 4 or 3. So choose which one has the larger sum, in this case set the current value to 3.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3								
global	4									

4.3. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value does not change.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3								
global	4	4								

4.4. Next is index 2, identify the maximum sub-array ending at this index. It is either ***sub-array [2]*** which is equal to 5, ***sub-array [1, 2]*** which is equal to -1 + 5 or 4, or ***sub-array [0, 1, 2]*** which is equal to 4 – 1 + 5 or 8. So choose which one has the larger sum, in this case set the current value to 8.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3	8							
global	4	4								

4.5. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value is set to 8.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3	8							
global	4	4	8							

4.6. Next is index 3, identify the maximum sub-array ending at this index. It is either ***sub-array [3]*** which is equal to -4, ***sub-array [2, 3]*** which is equal to 5 + -4 or 1, ***sub-array [1, 2, 3]*** which is equal to -1 + 5 -4 or 0, or ***sub-array [0, 1, 2, 3]*** which is equal to 4 – 1 + 5 -4 or 4. So choose which one has the larger sum, **in this case set the current value to 4**.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3	8	4						
global	4	4	8							

4.7. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value does not change.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3	8	4						
global	4	4	8	8						

4.8. Next is index 4, identify the maximum sub-array ending at this index. It is either ***sub-array [4]*** which is equal to 2, ***sub-array [3, 4]*** which is equal to -4 + 2 or -2, ***sub-array [2, 3, 4]*** which is equal to 5 - 4 + 2 or 3, ***sub-array [1, 2, 3, 4]*** which is equal to – 1 + 5 - 4 + 2 or 2, or ***sub-array [0, 1, 2, 3, 4]*** which is equal to 4 – 1 + 5 – 4 + 2 or 6. So choose which one has the larger sum, **in this case set the current value to 6**.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3	8	4	6					
global	4	4	8	8						

4.9. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value does not change.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	**4**	**3**	**8**	**4**	**6**					
global	**4**	**4**	**8**	**8**	**8**					

4.10. Next is index 5, identify the maximum sub-array ending at this index. It is either ***sub-array [5]*** which is equal to 6, ***sub-array [4, 5]*** which is equal to 2 + 6 or 8, ***sub-array [3, 4, 5]*** which is equal to -4 + 2 + 6 or 4, ***sub-array [2, 3, 4, 5]*** which is equal to 5 - 4 + 2 + 6 or 9, ***sub-array [1, 2, 3, 4, 5]*** which is equal to – 1 + 5 – 4 + 2 + 6 or 8, or ***sub-array [0, 1, 2, 3, 4, 5]*** which is equal to 4 – 1 + 5 – 4 + 2 + 6 or 12. So choose which one has the larger sum, **in this case set the current value to 12**.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	**4**	**3**	**8**	**4**	**6**	**12**				
global	**4**	**4**	**8**	**8**	**8**					

4.11. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value is set to 12.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	**4**	**3**	**8**	**4**	**6**	**12**				
global	**4**	**4**	**8**	**8**	**8**	**12**				

4.12. Next is index 6, identify the maximum sub-array ending at this index. It is either ***sub-array [6]*** which is equal to -8, ***sub-array [5, 6]*** which is equal to 6 – 8 or -2, ***sub-array [4, 5, 6]*** which is equal to 2 + 6 - 8 or 0, ***sub-array [3, 4, 5, 6]*** which is equal to - 4 + 2 + 6 – 8 or -4, ***sub-array [2, 3, 4, 5, 6]*** which is equal to 5 – 4 + 2 + 6 - 8 or 1, ***sub-array [1, 2, 3, 4, 5, 6]*** which is equal to – 1 + 5 – 4 + 2 + 6 – 8 or 0, or ***sub-array [0, 1, 2, 3, 4, 5, 6]*** which is equal to 4 – 1 + 5 – 4 + 2 + 6 – 8 or 4. So choose which one has the larger sum, **in this case set the current value to 4**.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	**4**	**3**	**8**	**4**	**6**	**12**	**4**			
global	**4**	**4**	**8**	**8**	**8**	**12**				

4.13. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value does not change.

i	**[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**
current	**4**	**3**	**8**	**4**	**6**	**12**	**4**			
global	**4**	**4**	**8**	**8**	**8**	**12**	**12**			

4.14. Next is index 7, identify the maximum sub-array ending at this index. It is either ***sub-array [7]*** which is equal to -2, ***sub-array [6, 7]*** which is equal to – 8 – 2 or -10, ***sub-array [5, 6, 7]*** which is equal to 6 - 8 – 2 or -2, ***sub-array [4, 5, 6, 7]*** which is equal to 2 + 6 – 8 – 2 or -2, ***sub-array [3, 4, 5, 6, 7]*** which is equal to – 4 + 2 + 6 – 8 – 2 or -6, ***sub-array [2, 3, 4, 5, 6, 7]*** which is equal to 5 – 4 + 2 + 6 – 8 – 2 or -1, ***sub-array [1, 2, 3, 4, 5, 6, 7]*** which is equal to – 1 + 5 – 4 + 2 + 6 – 8 – 2 or -2, or ***sub-array [0, 1, 2, 3, 4, 5, 6, 7]*** which is equal to 4 – 1 + 5 - 4 + 2 + 6 -8 – 2 or 2. So choose which one has the larger sum, **in this case set the current value to 2**.

i	**[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**
current	**4**	**3**	**8**	**4**	**6**	**12**	**4**	**2**		
global	**4**	**4**	**8**	**8**	**8**	**12**	**12**			

4.15. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value does not change.

i	**[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**
current	**4**	**3**	**8**	**4**	**6**	**12**	**4**	**2**		
global	**4**	**4**	**8**	**8**	**8**	**12**	**12**	**12**		

4.16. Next is index 8, identify the maximum sub-array ending at this index. It is either ***sub-array [8]*** which is equal to 9, ***sub-array [7, 8]*** which is equal to – 2 + 9 or 7, ***sub-array [6, 7, 8]*** which is equal to - 8 – 2 + 9 or -1, ***sub-array [5, 6, 7, 8]*** which is equal to 6 – 8 – 2 + 9 or 5, ***sub-array [4, 5, 6, 7, 8]*** which is equal to - 2 + 6 – 8 – 2 + 9 or 3, ***sub-array [3, 4, 5, 6, 7, 8]*** which is equal to - 4 + 2 + 6 – 8 – 2 + 9 or 3, ***sub-array [2, 3, 4, 5, 6, 7, 8]*** which is equal to 5 – 4 + 2 + 6 – 8 – 2 + 9 or 8, ***sub-array [1, 2, 3, 4, 5, 6, 7, 8]*** which is equal to – 1 + 5 - 4 + 2 + 6 -8 – 2 + 9 or 7, or ***sub-array [0, 1, 2, 3, 4, 5, 6, 7, 8]*** which is equal to 4 – 1 + 5 – 4 + 2 – 8 – 2 + 9 or 11. So choose which one has the larger sum, **in this case set the current value to 11.**

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	**4**	**3**	**8**	**4**	**6**	**12**	**4**	**2**	**11**	
global	**4**	**4**	**8**	**8**	**8**	**12**	**12**	**12**		

4.17. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value does not change.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	**4**	**3**	**8**	**4**	**6**	**12**	**4**	**2**	**11**	
global	**4**	**4**	**8**	**8**	**8**	**12**	**12**	**12**	**12**	

4.18. Next is index 9, identify the maximum sub-array ending at this index. It is either ***sub-array [9]*** which is equal to -3, ***sub-array [8, 9]*** which is equal to 9 – 3 or 6, ***sub-array [7, 8, 9]*** which is equal to - 2 + 9 – 3 or 4, ***sub-array [6, 7, 8, 9]*** which is equal to – 8 – 2 + 9 – 3 or -4, ***sub-array [5, 6, 7, 8, 9]*** which is equal to 6 – 8 – 2 + 9 – 3 or 2, ***sub-array [4, 5, 6, 7, 8, 9]*** which is equal to 2 + 6 – 8 – 2 + 9 – 2 or 5, ***sub-array [3, 4, 5, 6, 7, 8, 9]*** which is equal to – 4 + 2 + 6 – 8 – 2 + 9 – 3 or 0, ***sub-array [2, 3, 4, 5, 6, 7, 8, 9]*** which is equal to 5 - 4 + 2 + 6 -8 – 2 + 9 – 3 or 5, ***sub-array [1, 2, 3, 4, 5, 6, 7, 8, 9]*** which is equal to – 1 + 5 – 4 + 2 – 8 – 2 + 9 – 3 or -2, or ***sub-array [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]*** which is equal to 4 – 1 + 5 – 4 + 2 + 6 – 8 – 2 + 9 – 3 or 8. So choose which one has the larger sum, **in this case set the current value to 8.**

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	**4**	**3**	**8**	**4**	**6**	**12**	**4**	**2**	**11**	**8**
global	**4**	**4**	**8**	**8**	**8**	**12**	**12**	**12**	**12**	

4.19. Then compare the current and global variable, **if the current value is greater than the global value, then change the global value to the current value**. Otherwise, **if the current value is less than the global value then it remains the same.** In this case, the global value does not change.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3	8	4	6	12	4	2	11	8
global	4	4	8	8	8	12	12	12	12	12

4.20. The maximum prize to win in riding the busses is 12. In order to get the max price you must start boarding and boarding on the first stop A[0] up to the fifth stop A[5].

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
current	4	3	8	4	6	12	4	2	11	8
global	4	4	8	8	8	12	12	12	12	12

Bus stops you should board / deboard:
Bus Stop #1: 4 (A[0])
Bus Stop #2: -1 (A[1])
Bus Stop #3: 5 (A[2])
Bus Stop #4: -4 (A[3])
Bus Stop #5: 2 (A[4])
Bus Stop #6: 6 (A[5])

Total prize to get: 12