Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

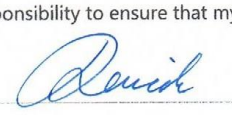
Last name:	DAO	Student ID:	17726966
Other name(s):	DERICK VIET		
Unit name:	DESIGN AND ANALYSIS OF ALGORITHMS	Unit ID:	COMP 3001
Lecturer / unit coordinator:	SIE TENG SOH	Tutor:	SIE TENG SOH
Date of submission:	15 MAY 2017	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature:  Date of signature: 15/05/17

(By submitting this form, you indicate that you agree with all the above text.)

Contents

Introduction

This report answers the questions supplied in the assignment specifications for Design and Analysis of Algorithms (COMP3001). The report features the use of the Master method, analysis of time complexity, creation of algorithms and sorting. The questions are made as headers to guide for easy and clearly provided answers which should summarise all or if not, most of the content that was seen to be relevant to include in discussing the various questions.

Question One (Total: 20 marks)

a) ***(10 marks).*** *Use the Master method to solve the following recurrence function:*

$$T(n) = 4T(\sqrt{n}) + \log_2^2 n$$

Note: *The function is not in a form suitable for the master method. However, you can convert it to the required form for the method by using another variable $k = \log_2 n$.*

$T(n) = 4T(\sqrt{n}) + \log_2^2 n$
$k = \log_2 n$, $n = 2^k$
$T(2^k) = 4(2^{k/2}) + k^2$
$S(k) = T(2^k) = 4[S(k/2)] + k^2$

1. $a = 4$, $b = 2$, $f(k) = k^2$
2. $\log_b a = \log_2 4 = 2 \rightarrow k^2$
3. $k^2 \geq f(k)$, $k^2 = k^2$. Therefore, CASE 2

Therefore, time complexity of $S(k) = O(k^2 \log_2 k)$.
$S(k) = T(2^k)$, therefore $T(2^k) = O(k^2 \log_2 k)$.
$T(n) = O(\log_2^2 n \cdot \log_2(\log_2 n))$.

b) ***(10 marks).*** *Formally prove or disprove the correctness of each of the followings. Note that you must show the values of c and n_0 in your proof.*

a. *$(n + a)^b = \Theta(n^b)$, for any real constants a and b, and for $a > 0$, $b > 0$.*

$(n + a)^b = \Theta(n^b)$

PROVE $(n + a)^b \leq O(n^b)$
$(n + a)^b \leq c_1 \cdot n^b$
$(n + a)^b \leq (n + n)^b$ for $n \geq a$
$(n + n)^b \leq c_1 \cdot n^b$
$(2n)^b \leq c_1 \cdot n^b$
$2^b \cdot n^b \leq c_1 \cdot n^b$
$c_1 \geq 2^b$

Therefore, $(n + a)^b \leq O(n^b)$ is true for values, $c_1 \geq 2^b$ and $n_0 \geq a$.

PROVE $(n + a)^b \geq \Omega(n^b)$
$(n + a)^b \geq c_2 \cdot n^b$
$(n + a)^b \geq n^b$ for $n \geq 0$

$n^b \geq c_2 \cdot n^b$
$c_2 \leq 1$

Therefore, $(n + a)^b \geq \Omega(n^b)$ is true for values, $c_2 \leq 1$ and $n_0 \geq 0$.

Therefore, $(n + a)^b = \Theta(n^b)$ is true for values, $c_1 \geq 2^b$, $c_2 \leq 1$ and $n_0 \geq a$.

b. $2^{n+1} = O(2^n)$

PROVE $2^{n+1} = O(2^n)$
$2^{n+1} \leq c_1 \cdot 2^n$
$2^{n+1} \leq 2 \cdot 2^n$ for $n \geq 0$
$2 \cdot 2^n \leq c_1 \cdot 2^n$
$c_1 \geq 2$

Therefore, $2^{n+1} = O(2^n)$ is true for values, $c_1 \geq 2$ and $n \geq 0$.

c. $2^{2n} = O(2^n)$

PROVE $2^{2n} \leq c_1 \cdot 2^n$
$2^{2n} \leq 2^n \cdot 2^n$ for $n \geq 0$
$2^n \cdot 2^n \leq c_1 \cdot 2^n$

Given $n \geq 0$, let $n = 0$. This will allow for $c_1 = 1$.
Then, $2^n \cdot 2^n \leq 1 \cdot 2^n$, for $n \geq 0$.
Now, let $n = 1$.
$4 \leq 2$. This is not true and in result, dictates our previous range for n ($n \geq 0$) to be invalid.
As n increases, c_1 stays constant and 2^n gets bigger and eventually will be bigger than c_1.

In a general case, Let $c_1 = 2^k$, a constant number of 2 to the power of an integer k.
Let $n = k + 1$, $2^{k+1} \cdot 2^{k+1} \leq 2^k \cdot 2^{k+1}$.
The left side of the equation for any k will always be bigger than constant c_1 which is 2^k.
Hence, $2^{2n} = O(2^n)$ has been disproven.

Question Two (Total: 40 marks)

a) ***(20 marks).*** *Design an algorithm to generate a (p x q) sorted grid G from a given list L of n integers, i.e., solve* ***Step 1****.*

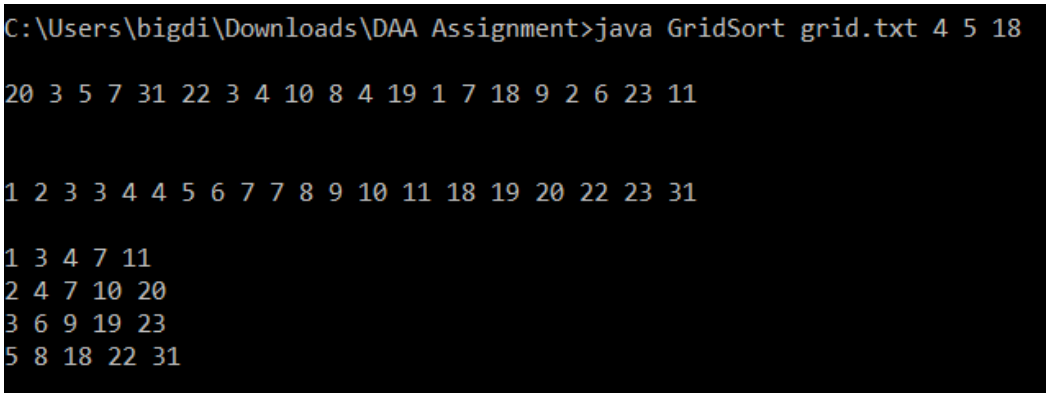
```
createGrid(values, row, col)
IMPORT: int[] values, int row, int col
EXPORT: grid

r ← 0
c ← 0
lastRow ← 0
lastCol ← 0
count ← 0
grid = create new int[row][col]

/* USE MERGESORT ON THE GIVEN LIST (CHECK APPENDIX FOR PSEUDOCODE) */

WHILE r ≤ row AND c < col
    IF lastRow ≤ row
        lastRow ← r + 1
    IF lastCol != col
        lastCol ← c

    WHILE c < col AND r ≥ 0
        grid[r][c] ← values[count]
        count++
        r--
        c++
    ENDWHILE
    r ← lastRow
    IF r = row
        r--
        c ← lastCol
        c++
    ELSE
        c = 0
ENDWHILE

return grid
```

a. *Briefly explain why your algorithm is correct.*

My algorithm is correct because I utilize the MergeSort sorting algorithm to sort my list of values into ascending order and insert them diagonally from the top left hand corner of the grid index to the bottom right index, with each iteration filling up spaces of their respective diagonal rows to create a sorted grid that is in increasing order for every row

and every column.

After implementing my code in Java, I can conclude that my algorithm is correct. The output that was printed to the command terminal when sorting the list L of 20 values creates a 'sorted grid' or in technical terms, a 'young tableau' which is a grid that has its rows and columns in ascending order.



Following the pseudo code that was implemented for creating the grid (CREATE_GRID), after sorting the list of values, for debugging purposes, I printed out the list of values before and after it is sorted to distinguish that it was sorted by using MergeSort. The first line of values is the list that was provided in the specification for this question unsorted, and when sorted using MergeSort, it produces the second line of values. This shows that my sorting algorithm works.

After sorting my list of values, I use the remaining sections of code from CREATE_GRID to insert elements of the list of values into a (4 x 5) grid to create a sorted grid or a young tableau. As shown in the image, the table that has been printed out onto the terminal shows the values at each index in the 2D array to show that insertion into a grid was successful and is correct to how the specifications indicated, to create a sorted (4 x 5) grid.

b. Analyse the time complexity of your algorithm.

```
createGrid(values, row, col)
IMPORT: int[] values, int row, int col
EXPORT: grid

r ← 0
c ← 0
lastRow ← 0
lastCol ← 0
count ← 0
grid = create new int[row][col]

/* USE MERGESORT ON THE LIST values (CHECK APPENDIX FOR PSEUDO CODE) */

WHILE r ≤ row AND c < col
    IF lastRow ≤ row
        lastRow ← r + 1
    IF lastCol != col
        lastCol ← c

    WHILE r ≥ 0 AND c < col
        grid[r][c] ← values[count]
        count++
        r--
        c++
    ENDWHILE
    r ← lastRow
    IF r = row
        r--
        c ← lastCol
        c++
    ELSE
        c = 0
ENDWHILE

return grid
```

O(nLog(n))

ALL O(1) Operations

O(1)

O(1)

O(1)

O(1)

ALL O(1) Operations

O(r) + O(c) times

O(1)

ALL O(1) Operations

O(1)

O(r) + O(c) times

Using Mergesort as a sorting algorithm to sort my list of values L will take time complexity of O(n log n)as analysed in the lectures. I am using MergeSort because the average case of the sort is the same as the sort's worst case and it's best case. Realistically thinking, there is a very unlikelihood that a list of values will be passed through as the best case for the sort. So to accommodate all possible situations, having a time complexity of O(n log n) as a worst case time complexity is the best way to use a particular sorting algorithm to sort my list of values to insert into the grid. MergeSort takes O(n log n) time, where n is the number of elements in the list.

The first WHILE loop runs at time complexity O(r) + O(c) times because r can range from

zero to the number of rows specified through the row variable imported into the function and c can range from zero to the number of columns likewise. The time complexity of this WHILE loop is O(r + c) time. The second WHILE loop runs at time complexity of O(r) + O(c) alike to the outer WHILE loop at the same speed, O(r + c). The inner WHILE runs at O(r) because variable r can ever reach maximum the range of the number of rows. 'r' ranges from number of rows to zero, decrementing each time the while loop is run. The variable 'c' can range from zero to the number of columns, hence the time complexity of c < col is O(c). Therefore, with the inner loop performing at O(r + c) and the outer loop performing at the same speed, the time complexity of the whole algorithm minus the MergeSort is $O((r+c)^2)$ where r is the number of rows and c is the number of columns.

In total, my whole CREATE_GRID function takes O(n log n) + $O((r+c)^2)$ time which totals to O(n log n + $(r+c)^2$) where n is the number of elements in the list of numbers to sort (in this case, its int[] values), r is the number of rows and c is the number of columns specified for the grid to be created in.

c. *Show how your algorithm produces the (4 x 5) sorted grid for the following list of n = 20 integers:*

L = <20, 3, 5, 7, 31, 22, 3, 4, 10, 8, 4, 19, 1, 7, 18, 9, 2, 6, 23, 11>

As explained above in section (a), my algorithm has created the (4 x 5) sorted grid asked in the specifications using the numbers specified above as input. The results of my Java coding allowed for this result when trying to create my sorted grid.

```
C:\Users\bigdi\Downloads\DAA Assignment>java GridSort grid.txt 4 5 18

20 3 5 7 31 22 3 4 10 8 4 19 1 7 18 9 2 6 23 11


1 2 3 3 4 4 5 6 7 7 8 9 10 11 18 19 20 22 23 31

1 3 4 7 11
2 4 7 10 20
3 6 9 19 23
5 8 18 22 31
```

The code below was the code I used to implement the creation of the sorted grid and with the use of print statements, it allowed me to show my results onto the command line.

```java
public static int[][] createGrid(int[] values, int row, int col)
{
    int r = 0, c = 0, lastRow = 0, lastCol = 0, count = 0;
    int grid[][] = new int[row][col];

    printList(values);
    System.out.println("");
    mergeSort(values);
    printList(values);

    while(r <= row && c < col)
    {
        if(lastRow <= row)
        {
            lastRow = r+1;
        }
        if(lastCol != col)
        {
            lastCol = c;
        }

        while(r >= 0 && c < col)
        {
            grid[r][c] = values[count];
            count++;
            r--;
            c++;
        }
        r = lastRow;
        if(r == row)
        {
            r--;
            c = lastCol;
            c++;
        }
        else
        {
            c = 0;
        }
    }
    return grid;
}
```

```java
public static void main(String[] args)
{
    int row = Integer.parseInt(args[0]);
    int col = Integer.parseInt(args[1]);
    int k = Integer.parseInt(args[2]);

    System.out.println("");


    int[] values = {20, 3, 5, 7, 31, 22, 3, 4, 10, 8, 4, 19, 1, 7, 18, 9, 2, 6, 23, 11};

    int[][] grid = createGrid(values, row, col);
    printGrid(grid, row, col);
}

public static void printGrid(int grid[][], int row, int col)
{
    for(int ii = 0; ii < row; ii++)
    {
        for(int jj = 0; jj < col; jj++)
        {
            System.out.print(grid[ii][jj] + " ");
        }
        System.out.println("");
    }
}

private static void printList(int[] list)
{
    for(int ii = 0; ii < 20; ii++)
    {
        System.out.print(list[ii] + " ");
    }
    System.out.println("\n");
}
```

In my main method, I have specified the grid dimensions as command line parameters and k as the value to find in my grid later on. The list of values are as stated in the specifications in an array of integers. A 2D grid is created by calling the createGrid() function, taking as its import, the list of values to sort, the row and column dimensions to create the grid. createGrid() creates a 2D grid from scratch and uses MergeSort to sort the elements in the list. After MergeSorting the list of values, createGrid() will insert the sorted array into a 2D grid and return it to main for later use. This shows that my algorithm has produced a (4 x 5) sorted grid by sorting using MergeSort and inserting them into a 2D array.

```java
public static void mergeSort(int[] A)
{
    mergeSortRecurse(A, 0, A.length-1);
}

private static void mergeSortRecurse(int[] A, int leftIdx, int rightIdx)
{
    int midIdx;
    if( leftIdx < rightIdx)
    {
        midIdx = (leftIdx + rightIdx) / 2;
        mergeSortRecurse(A, leftIdx, midIdx);
        mergeSortRecurse(A, midIdx + 1, rightIdx);
        merge(A, leftIdx, midIdx, rightIdx);
    }
}
```

```java
private static void merge(int[] A, int leftIdx, int midIdx, int rightIdx)
{
    int[] tempArr;
    int ii, jj, kk;
    tempArr = new int[rightIdx - leftIdx + 1];
    ii = leftIdx;
    jj = midIdx + 1;
    kk = 0;

    while( (ii <= midIdx) && (jj <= rightIdx))
    {
        if( A[ii] <= A[jj])
        {
            tempArr[kk] = A[ii];
            ii++;
        }
        else
        {
            tempArr[kk] = A[jj];
            jj++;
        }
        kk++;
    }

    for( ii = ii; ii < midIdx + 1; ii++)
    {
        tempArr[kk] = A[ii];
        kk++;
    }
    for( jj = jj; jj < rightIdx + 1; jj++)
    {
        tempArr[kk] = A[jj];
        kk++;
    }
    for( kk = leftIdx; kk < rightIdx + 1; kk++)
    {
        A[kk] = tempArr[kk - leftIdx];
    }
}
```

b) ***(20 marks).*** *Design an algorithm to search a key k from (p x q) sorted grid G in O(p + q) time, i.e., solve* ***Step 2****. More specifically, design an O(p + q)-time algorithm that, given as input a (p x q) sorted grid and an integer k, returns whether or not that integer is contained in the grid.*

a. *Your algorithm should use only O(1) additional space.*

```
search_key(grid, row, col, k)
IMPORT: int grid[][], int row, int col, int k
EXPORT: boolean

r ← 0
c ← col – 1

WHILE r < row AND c ≥ 0
        IF grid[r][c] = k
                return true
        ELSE IF grid[r][c] > k
                c ← c - 1
        ELSE
                r ← r + 1
ENDWHILE

return false
```

b. *Briefly explain why your algorithm is correct.*

The SEARCH_KEY method takes in a 2D grid which contains the values made from converting a list of values into a sorted grid, along with the row and column of the dimensions of the grid and an integer k which is needed to be searched for in the grid.

This algorithm allows for a grid with rows P and columns Q to be able to search for a certain value k. The searching method starts from the top right corner of the grid, the search traverses left if the number in the current position is greater than the key to find and down if the value is less than the number to find. This will always provide a new value in the grid closer to the number we are trying to find. As you move left, the number gets smaller and as you move down the grid, the number gets bigger. This will allow the algorithm to get closer and closer to the number depending on how big or small the number is initially at the start of the search with the value in comparison with the value in the top right hand corner. The results of my algorithm being correct is shown in my output on my terminal in part (d) of this question. It shows the various output statements and the steps that the algorithm took to search for a specific key.

c. Analyse the time complexity of your algorithm to show that its time complexity is O(p + q).

```
SEARCH_KEY
IMPORT: int grid[][], int row, int col, int k
EXPORT: boolean

r ← 0                                          O(1)
c ← col – 1                                    O(1)

WHILE r < row AND c ≥ 0                        O(row) + O(col) times
    IF grid[r][c] = k                          O(1)
        return true                            O(1)
    ELSE IF grid[r][c] > k                     O(1)
        c ← c - 1                              O(1)
    ELSE
        r ← r + 1                              O(1)
ENDWHILE

return false
```

All assignments and comparisons are O(1) time complexity except for the WHILE loop. The WHILE loop needs to check for r < row and needs to also check that col ≥ 0. The range that the value r can get to is the value of row which is imported into the function to specify the number of rows in the grid. The value of c starts from the number of columns (col) specified through the import parameter and can decrease to zero. The range of numbers c can hold is from zero to the number of columns – 1 due to arrays indexing at zero. Therefore, c can range from zero to the number of columns. Therefore, the WHILE loop needs to check two variables: r and c, in worst case, their max range of numbers where r can range from zero to the number of rows and c from zero to the number of columns. Due to the need of evaluating two separate variables, the WHILE loop is to run O(range of r) + O(range of c) times. Therefore, O(row) + O(column), which results with O(row + column) time complexity.

O(p + q) is a time complexity where p dictates the number of rows and q dictates the number of columns. Hence, as explained previously, O(row + column) matches with O(p + q) and therefore, my algorithm runs at O(p + q) time complexity.

d. Show how your algorithm works on the sorted grid that you generated in part a), for (i) k = 7, and (ii) k = 12.

My output to the terminal shows the steps the algorithm takes to search for a specified number k. For k = 7, my program has successfully found the number when traversing through the grid.

```
1 3 4 7 11
2 4 7 10 20
3 6 9 19 23
5 8 18 22 31

== SEARCHING GRID FOR K: 7 ==

CURRENT VALUE AT ROW: 0, COL: 4 is: 11
11 IS GREATER THAN K: 7 | MOVING LEFT

CURRENT VALUE AT ROW: 0, COL: 3 is: 7
FOUND K: 7

SUCCESFULLY FOUND K: 7

== ENDING SEARCH FOR K: 7 ==


C:\Users\bigdi\Downloads\DAA Assignment>
```

Given the grid that was sorted and inserted previously for part (a), the algorithm starts at the top right index (0,4) which is value of 11. That value is greater than 7, so the index it next moves to is the index to its left. That index holds the k value we are looking for, so the method returns a boolean saying that it has been successfully found and prints out a statement that it has been found and terminates the search.

For this question's (ii) to find k = 12, the search isn't able to find that value because it does not exist in the list of values to sort to begin with. The search goes through various values and gets to the end to state that the value has not been found.

```
1 3 4 7 11
2 4 7 10 20
3 6 9 19 23
5 8 18 22 31

== SEARCHING GRID FOR K: 12 ==

CURRENT VALUE AT ROW: 0, COL: 4 is: 11
11 IS LESS THAN K: 12 | MOVING DOWN

CURRENT VALUE AT ROW: 1, COL: 4 is: 20
20 IS GREATER THAN K: 12 | MOVING LEFT

CURRENT VALUE AT ROW: 1, COL: 3 is: 10
10 IS LESS THAN K: 12 | MOVING DOWN

CURRENT VALUE AT ROW: 2, COL: 3 is: 19
19 IS GREATER THAN K: 12 | MOVING LEFT

CURRENT VALUE AT ROW: 2, COL: 2 is: 9
9 IS LESS THAN K: 12 | MOVING DOWN

CURRENT VALUE AT ROW: 3, COL: 2 is: 18
18 IS GREATER THAN K: 12 | MOVING LEFT

CURRENT VALUE AT ROW: 3, COL: 1 is: 8
8 IS LESS THAN K: 12 | MOVING DOWN

K: 12 HAS NOT BEEN FOUND

== ENDING SEARCH FOR K: 12 ==


C:\Users\bigdi\Downloads\DAA Assignment>
```

For the algorithm to find k = 12, the algorithm starts with the top right corner's value.

The value 11 is less than 12 so the index shifts down by one. The new value is 20 which is greater than our k to find, the index moves left. The new index holds the value of 10 which is less than 12 so the index moves left. At the new index, the value is 19 and is greater than 12 which causes the index to move down. Resulting from that, the value of 9 is compared to 12. Nine being the lesser value, the index moves left to appear at 18 which is greater than 12. The index shifts for the last time moving down to 8. Due to 8 being lower than 12, the algorithm is caused to move down, but there is no longer any value below 8, so the search terminates, the method returns false and states that the number k = 12 has not been found.

These are the results from my executing of my algorithm to find the two numbers stated for this part of the question.

Question 3 (Total: 40 marks)

a) ***(20 marks).*** *Write a pseudocode of an efficient algorithm to generate 2DP/C for a given (s, t) pair. From the pseudocode, analyse the time complexity of the algorithm.*

You are allowed to use any available algorithm in the literature for the pseudocode. In this case, you must explicitly state the source.

```
dijkstra(startName)
IMPORT: String startName
EXPORT: none

IF startName IS NOT a key in the graph
        PRINT ERROR
        return;

source ← graph.get(startName)
q ← new TreeSet<Vertex>
```

ALL O(1) Operations

```
FOR EACH Vertex v in graph.values()
        IF v = source
                v.previous ← source
                v.dist = 0
        ELSE
                v.previous = null
                v.dist = INFINITY
        add v to q
```

O(V)

ALL O(1) Operations

```
dijkstra(q)
```

→ $O(V^2)$

```
djikstra(q)
IMPORT: set<Vertex> q
EXPORT: none

WHILE q IS NOT empty
        u ← q.pollFirst()
        IF u.dist = INFINITY
                break

        FOR EACH Map entries a IN u.neighbours.entrySet()
                v ← a.getKey()

                alternateDist ← u.dist() + a.getValue() + u.potential + v.potential
                IF alternateDist < v.dist
                        remove v from q
                        v.dist ← alternateDist
                        v.previous ← u
```

Runs O(V) times

ALL O(1) Operations

Runs O(V) times

ALL O(1) Operations

add v to q → O(1)

ENDWHILE

revEdges(endName)
IMPORT: String endName
EXPORT: none

v ← graph.get(endName) → O(1)

WHILE v.prev IS NOT itself AND v.previous IS NOT null — O(V)
    w ← v.previous
    weight ← v.neighbours.get(w)
    remove w from v's neighbour list
    remove v from w's neighbour list
    add w with -weight to v's neighbour list
    v ← w
ENDWHILE

All O(1) operations

potentials()
IMPORT: none
EXPORT: none

FOR EACH Vertices v in graph.values() — O(V)
    v.potential ← v.dist

resPath(String endName)
IMPORT: String endName
EXPORT: none

v ← graph.get(endName) → O(1)

WHILE v.previous IS NOT itself AND v.previous IS NOT null — O(V)
    from ← v.previous.name
    to ← v.name
    IF edge exists in answer tree from 'from' - 'to' with distance of 0 → O(E)
        remove that edge
    ELSE
        add that new edge
    v ← v.previous

main()
dijkstra(START) → $O(V^3)$
resPath(END) → O(VE)
revEdges(END) → O(V)
potentials() ← O(V)
djikstra(START) → $O(V^3)$
resPath(END) ← O(VE)

Starting with the function dijkstra with a String import has a FOR loop which goes through all vertices in graph.values() and compares each of them to the source. This function takes O(V) time complexity where V is the number of vertices in the graph.values() structure. Additionally, the function will call on dijkstra() again but with the set import. This function utilizes a WHILE loop which loops through until q is not empty of vertices. Within that WHILE loop is a FOR EACH loop which loops for all map entries in the neighbour's set of vertices to assign a vertex their own distance to the goal node by using an estimated alternateDist value. With that, this function performs at $O(V^2)$ speed where V is the number of vertices in the neighbouring set and additionally, the number of vertices in the set q. Therefore, totaling with the initial time complexity of dijkstra with a String as an import, the total time complexity of that function is $O(V^3)$.

The function revEdges reverts the edges of vertices from the end node back to a specific node. The WHILE loop loops from one vertex back to V numbers of vertices in its path or that connects to the one starting vertex. This function therefore performs at time complexity O(V) where V is the number of vertices that the function backtracks using v.previous. Function potentials() adjusts all vertices' potential value to their distance value to inform the potential of a node being closer to the goal. The FOR EACH loop loops for each vertex in graph.values() and adjusts each of their potential values. This takes O(V) times where V is the number of vertices in set of vertices supplied by graph.values().

The method resPath restore the path from the end node and backtracks. The WHILE loop functions until all vertices in the chain of previous vertices from the initial node are at the end of the path with no connecting link. In turn, this function takes O(V) time to function where V is the number of vertices the function has to backtrack until it gets to a node with a previous node of null. Additionally, the IF statement to check whether an edge is part of the answer tree will search through E edges to find if the edge is existent in the tree. Thus, the method's total time complexity would be O(V)*O(E) which equals to a time complexity of O(VE).

Concluding, main() executes the whole algorithm to find the 2DP/C by executing dijkstra, resPath, revEdges and potentials in the order that was specified above. The time that is taken in total to execute this program is the time complexity of each of those function calls. The total counts to: $O(V^3)$ (djikstra) + O(VE) (resPath) + O(V) (revEdges) + O(V) (potentials) + $O(V^3)$ (djikstra) + O(VE) (resPath) = $O(V^3)$ because $V^2 \geq E$, where the number of vertices squared is always greater than the number of edges.

*b) **(20 marks).** Use the algorithm in part (a) to generate the 2DP/C for each of the following (s, t) pair of the given graph. **You have to show the details of your solution.***

a. (C, E)

```
====== EXECUTING DIJKSTRA ======

VERTEX: A IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: B IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: C IS THE SOURCE: C
CHANGING PREVIOUS OF V TO SOURCE AND IT'S DISTANCE TO 0

VERTEX: D IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: E IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: F IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: G IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

CHANGING DISTANCE OF F TO ALTERNATE DISTANCE: 5
CHANGING DISTANCE OF A TO ALTERNATE DISTANCE: 7
CHANGING DISTANCE OF G TO ALTERNATE DISTANCE: 16
CHANGING DISTANCE OF D TO ALTERNATE DISTANCE: 9
CHANGING DISTANCE OF B TO ALTERNATE DISTANCE: 13
CHANGING DISTANCE OF E TO ALTERNATE DISTANCE: 16
CHANGING DISTANCE OF G TO ALTERNATE DISTANCE: 12
```

```
====== EXECUTING DIJKSTRA AGAIN ======

VERTEX: A IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: B IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: C IS THE SOURCE: C
CHANGING PREVIOUS OF V TO SOURCE AND IT'S DISTANCE TO 0

VERTEX: D IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: E IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: F IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: G IS NOT THE SOURCE: C
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

CHANGING DISTANCE OF A TO ALTERNATE DISTANCE: 0
CHANGING DISTANCE OF F TO ALTERNATE DISTANCE: 10
CHANGING DISTANCE OF B TO ALTERNATE DISTANCE: 0
CHANGING DISTANCE OF D TO ALTERNATE DISTANCE: 0
CHANGING DISTANCE OF E TO ALTERNATE DISTANCE: 6
CHANGING DISTANCE OF G TO ALTERNATE DISTANCE: 0
CHANGING DISTANCE OF F TO ALTERNATE DISTANCE: 0
CHANGING DISTANCE OF E TO ALTERNATE DISTANCE: 2
```

```
====== EXECUTING RESPATH ======

RESTORING PATH FROM: E


====== EXECUTING REVEDGES ======

REVERTING EDGES FROM END NODE: E


====== EXECUTING POTENTIALS ======

CHANGING ALL POTENTIAL OF ALL VERTICES WITH ITS DISTANCE!
```

```
====== EXECUTING RESPATH AGAIN ======

RESTORING PATH FROM: E

------------------------------------
2DP/C SOLUTIONS:

C -> A -> D -> E
C -> F -> D -> G -> E
------------------------------------

C:\Users\bigdi\Downloads\DAA Assignment\q3>
```

Executing dijkstra() the first time finds the starting node and changes all the the nodes that it visits with an alternate distance value. C chooses F which can expand to G and D and calculates those values. Then the algorithm picks A which is the next lowest score and expands to B. The algorithm then chooses D which expands to G again. Djikstra calculates the shortest path costs to various nodes from an initial node. After executing dijkstra, resPath, revEdges and potentials all execute their methods in respective order each performing what they have printed out to the command line. The algorithm then

executes djikstra again to find the cost of paths to various nodes to have the end node containing two nodes which create two separate disjoint paths to output to the screen. This shows the way my algorithm works when using Dijkstra's algorithm to perform two edge-disjoint paths with minimum cost from node C to node E. The result of this test is C → A -> D → E and C → F → D → G → E.

b. (F, E)

```
====== EXECUTING DIJKSTRA ======

VERTEX: A IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: B IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: C IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: D IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: E IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: F IS THE SOURCE: F
CHANGING PREVIOUS OF V TO SOURCE AND IT'S DISTANCE TO 0

VERTEX: G IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

CHANGING DISTANCE OF C TO ALTERNATE DISTANCE: 5
CHANGING DISTANCE OF G TO ALTERNATE DISTANCE: 11
CHANGING DISTANCE OF A TO ALTERNATE DISTANCE: 8
CHANGING DISTANCE OF D TO ALTERNATE DISTANCE: 4
CHANGING DISTANCE OF E TO ALTERNATE DISTANCE: 11
CHANGING DISTANCE OF G TO ALTERNATE DISTANCE: 7
CHANGING DISTANCE OF B TO ALTERNATE DISTANCE: 12
CHANGING DISTANCE OF A TO ALTERNATE DISTANCE: 6
```

```
====== EXECUTING DIJKSTRA AGAIN ======

VERTEX: A IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: B IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: C IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: D IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: E IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

VERTEX: F IS THE SOURCE: F
CHANGING PREVIOUS OF V TO SOURCE AND IT'S DISTANCE TO 0

VERTEX: G IS NOT THE SOURCE: F
CHANGING PREVIOUS OF V TO NULL AND IT'S DISTANCE TO INFINITY

CHANGING DISTANCE OF C TO ALTERNATE DISTANCE: 0
CHANGING DISTANCE OF G TO ALTERNATE DISTANCE: 4
CHANGING DISTANCE OF A TO ALTERNATE DISTANCE: 2
CHANGING DISTANCE OF B TO ALTERNATE DISTANCE: 2
CHANGING DISTANCE OF D TO ALTERNATE DISTANCE: 6
CHANGING DISTANCE OF E TO ALTERNATE DISTANCE: 12
CHANGING DISTANCE OF E TO ALTERNATE DISTANCE: 6
```

```
====== EXECUTING RESPATH ======

RESTORING PATH FROM: E


====== EXECUTING REVEDGES ======

REVERTING EDGES FROM END NODE: E


====== EXECUTING POTENTIALS ======

CHANGING ALL POTENTIAL OF ALL VERTICES WITH ITS DISTANCE!
```

```
====== EXECUTING RESPATH AGAIN ======

RESTORING PATH FROM: E

-----------------------------------
2DP/C SOLUTIONS:

F -> D -> E
F -> G -> E
-----------------------------------

C:\Users\bigdi\Downloads\DAA Assignment\q3>
```

Continuing on from the previous example of the two edge-disjoint paths given in part (a), this execution of the algorithm is finding the two paths that have minimum cost connecting node F to node E. The algorithm runs through and expands nodes with the smallest alternate distance value to be able to find the most suitable paths to the goal. F can expand to C, G, A and D. The algorithm picks D next which can expand to E, G or B. The next node the algorithm picks is the first expanded node C with value of 5 and

expands to node A. The algorithm runs resPath, revEdges and potential to update values of each vertex, reverting and restoring edges and paths from the finishing node E. The function djikstra() is executed again which in result, updates the finishing node E with two neighbouring nodes to create paths towards the starting node to supply a result of two edge-disjoint paths with minimum cost. The results of this test was: F → D → E and F → G → E.

Appendix

MergeSort Pseudocode

```
mergeSort(A)
IMPORT: int[] A
EXPORT: none

        mergeSortRecurse(A, 0, A.length – 1)

mergeSortRecurse(A, leftIdx, rightIdx)
IMPORT: int[] A, int leftIdx, int rightIdx
EXPORT: none

        IF leftIdx < rightIdx
                midIdx ← (leftIdx + rightIdx)/2
                mergeSortRecurse(A, leftIdx, midIdx)
                mergeSortRecurse(A, midIdx + 1, rightIdx)
                merge(A, leftIdx, midIdx, rightIdx);

merge(A, leftIdx, midIdx, rightIdx)
IMPORT: int[] A, int leftIdx, int midIdx, int rightIdx
EXPORT: none

        tempArr ← create new int[rightIdx – leftIdx + 1]
        ii ← leftIdx
        jj ← midIdx + 1
        kk ← 0

        WHILE ii ≤ midIdx AND jj ≤ rightIdx
                IF A[ii] ≤ A[jj]
                        tempArr[kk] ← A[ii]
                        ii++
                ELSE
                        tempArr[kk] ← A[jj]
                        jj++
                kk++
        ENDWHILE

        FOR ii to midIdx + 1, INC BY 1
                tempArr[kk] ← A[ii]
                kk++
        ENDFOR

        FOR jj to rightIdx, INC BY 1
                tempArr[kk] ← A[jj]
                kk++
        ENDFOR

        FOR kk ← leftIdx to rightIdx, INC BY 1
                A[kk] ← tempArr[kk – leftIdx]
        ENDFOR
```