Contents

1 Basic Test Results

```
Running...
Opening tar file
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK
Compilation check...
Compiling...
OK
Compilation seems OK! Check if you got warnings!

=====================
 Public test cases
=====================

=====================
Test #1
Running RailWayPlanner
OK
Running diff
OK
Test 1 passed.
=====================

=====================
Test #2
Running RailWayPlanner
OK
Running diff
OK
Test 2 passed.
=====================

=====================
Test #3
Running RailWayPlanner
OK
Running diff
OK
Test 3 passed.
=====================

=====================
Test #4
Running RailWayPlanner
OK
Running diff
OK
Test 4 passed.
=====================

=====================
Test #5
Running RailWayPlanner
OK
Running diff
OK
```

```
Test 5 passed.
====================

====================
Test #6
Running RailWayPlanner
OK
Running diff
OK
Test 6 passed.
====================

====================
Test #7
Running RailWayPlanner
OK
Running diff
OK
Test 7 passed.
====================

====================
Test #8
Running RailWayPlanner
OK
Running diff
OK
Test 8 passed.
====================

====================
Test #9
Running RailWayPlanner
OK
Running diff
OK
Test 9 passed.
====================

====================
Test #10
Running RailWayPlanner
OK
Running diff
OK
Test 10 passed.
====================

***********************************
*                                 *
*   presubmission script passed   *
*       10/10 tests passed        *
*                                 *
***********************************

========================
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 RailWayPlanner.c

```c
/**
 * @file RailWayPlanner.c
 * @author  Muaz Abdeen <muaz.abdeen@mail.huji.ac.il>
 * @ID 300575297
 * @date 7 May 2020
 *
 * @brief Program that receives from user a file contains rail details:
 *              - Rail length
 *              - Number of rail joints
 *              - Kinds of joints
 *              - Parts of rail <start joint, end joint, length, price>
 *            and given a valid input it calculates the minimal cost to build a rail of the
 *            given length from the given parts.
 *
 * @section DESCRIPTION
 * Program that calculates the minimal cost of the rail.
 * Input  : Rail's info: <length>,<number of joints>,<kins of joints>,<building parts>
 * Process: given a valid input it calculates the minimal cost to build a rail of the
 *              given length from the given parts.
 * Output : > If the input is not valid - print informative message.
 *           > Else if the input is valid - print the minimal cost.
 */

// ------------------------------ includes --------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <ctype.h>
#include <regex.h>
#include <stdbool.h>

// ------------------------------ macros & constants --------------------------------

#define MAX_ROW 1024
#define NUM_PART_DETAILS 4
#define INITIAL_ALLOC 10
#define NO_NUM -5
#define NOT_POSSIBLE INT_MAX
#define NO_INDEX -1
#define CANNOT_BUILD -1
#define OUTPUT_FILE "./railway_planner_output.txt"
#define ARGC_ERROR "Usage: RailWayPlanner <InputFile>"
#define NO_FILE_ERROR "File doesn't exists."
#define EMPTY_FILE_ERROR "File is empty."
#define INVALID_INPUT_ERROR "Invalid input in line: %d."
#define MIN_PRICE_RES "The minimal price is: %d"

// ------------------------------ functions & structs ---------------------------

typedef struct railPart railPart;
typedef struct algorithmInput AlgIn;

void outputMessage(char *arr, int num);
bool emptyFile(FILE *file);
bool validFile(FILE *filename);
bool checkDigit(char arr[]);
int checkInteger(char arr[]);
```

```c
bool checkJoints(char arr[], int numJoints);
char *jointsArray(char arr[], int numJoints);
bool helperCheckPartDetails(char detail[], int idx, char joints[]);
bool addPartDetail(char **details, AlgIn *railInfo, int *capacity, int line);
int checkDetails(FILE *file, AlgIn *railInfo);
int jointIDX(char joint, AlgIn railInfo);
int min(char joint, int row, const int table[], AlgIn railInfo);
int *tableBuilder(int rows, int columns, AlgIn railInfo);
int minTotalCost(const int *table, AlgIn railInfo);
void printTable(const int *table, AlgIn railInfo);  // Extra function to display the table
void freeAll(int **table, AlgIn *railInfo);

// ------------------------------------------------------------------------------

/**
 * @brief A structure to represent rail part details.
 */
struct railPart
{
    char start, end;   /**< starting and ending joints  */
    int length, price;
};


/**
 * @brief A structure to represent the rail info which received from input file.
 */
struct algorithmInput
{
    int railLen, numJoints, numParts;
    char *kindsJoints;   /**< different kinds of joints  */
    railPart *partsCollection;   /**< collection of all parts to be used in rail construction  */
};


/**
 * @brief A function to print a suitable message to output file.
 * @param arr C string represents the message
 * @param num number of line (in case it exists)
 */
void outputMessage(char *arr, int num)
{
    FILE *outputFile = fopen(OUTPUT_FILE, "w");
    if (outputFile == NULL)
    {
        exit(EXIT_FAILURE);
    }
    if (num == NO_NUM)   // the message doesn't contains a line number
    {
        fprintf(outputFile, "%s", arr);
    }
    else
    {
        fprintf(outputFile, arr, num);
    }
    fclose(outputFile);
}


/**
 * @brief A function to check if input file is empty.
 * @param file the input file.
 * @return true if empty, else, false.
 */
bool emptyFile(FILE *file)
{
    fseek(file, 0, SEEK_END);
    if (ftell(file) == 0)
    {
        return true;
    }
    rewind(file);   // return the pointer to the start
    return false;
}

/**
 * @brief A function to check if input file is exists and not empty.
 * @param file input file.
 * @return true if valid, else, false.
 */
bool validFile(FILE *file)
{
    if (file == NULL)   // fopen in main returns NULL
    {
        outputMessage(NO_FILE_ERROR, NO_NUM);
        return false;
    }
    if (emptyFile(file))
    {
        outputMessage(EMPTY_FILE_ERROR, NO_NUM);
        return false;
    }
    return true;
}

/**
 * @brief A function to check if chars in string are numbers only.
 * @param arr string supposed to represent a number.
 * @return true if it contains only numbers, else, false.
 */
bool checkDigit(char arr[])
{
    char *cutInput = (char*)malloc((int) strlen(arr) * sizeof(char) + 1);
    sscanf(arr, "%[^'\n']", cutInput);   // cut the newline char
    for (int i = 0; i < (int) strlen(cutInput); i++)
    {
        if (! isdigit(arr[i]))
        {
            return false;
        }
    }
    free(cutInput);
//    cutInput = NULL;
    return true;
}

/**
 * @brief A function to check and convert string to integer.
 * @param arr string supposed to represent a number.
 * @return the integer that represented in the string, ot -1 if it fails.
 */
int checkInteger(char arr[])
{
    if (! checkDigit(arr))
    {
        return -1;
    }
    char *ptr;
    int num = (int) strtol(arr, &ptr, 10);
    return num;
}

/**
 * @brief checks validity of joints in input file.
 * @param arr string contains all joints.
 * @param numJoints number of joints.
 * @return false if any of thw joints is more than one char, or their number exceeds
 *         the provided number in input file. else returns true.
 */
bool checkJoints(char arr[], int numJoints)
{
    // check if # of joint symbols exceeds # of joint kinds
    int count = 0;
    for (int i = 0; i < (int) strlen(arr) - 1; i++)
    {
        if (arr[i] == ',')
        {
            count++;
        }
    }

    if (count != (numJoints - 1))
    {
        return false;
    }

    // check if there is joint symbols with more than one char
    int jointSymbols = (int) strlen(arr) - count - 1;
    if (jointSymbols != numJoints)
    {
        return false;
    }
    return true;
}

/**
 * @brief builds a char array contains the joints.
 * @param arr string contains all joints separated by comma.
 * @param numJoints number of joints provided in input file.
 * @return the rail joints array.
 */
char *jointsArray(char arr[], int numJoints)
{
    // this dynamic array will be freed later in freeAll() function.
    char *railJoints = (char *)malloc(numJoints * sizeof(char) + 1);
    if (railJoints == NULL)
    {
        exit(EXIT_FAILURE);
    }
    char *ptr = strtok(arr, ",");
    for (int i = 0; i < numJoints; i++)
    {
        if (ptr != NULL)
        {
            railJoints[i] = *ptr;
            ptr = strtok(NULL, ",");
        }
    }
    return railJoints;
}

/**
 * @brief checks if a rail part is already existed in parts collection.
 * @param newPart a new part to be added to the collection.
 * @param existedPart an already existed part.
 * @return true if it is already existed, else, false.
 */
bool partExists(char *newPart[], railPart existedPart)
{
    char *lenPtr, *pricePtr;
    int partLen = (int) strtol(newPart[2], &lenPtr, 10);
    int partPrice = (int) strtol(newPart[3], &pricePtr, 10);

    if ((newPart[0][0] == existedPart.start) && (newPart[1][0] == existedPart.end) &&
```

```c
        (partLen == existedPart.length) && (partPrice == existedPart.price))
    {
        return true;
    }
    return false;
}

/**
 * @brief checks for every part if its details are valid
 * @param part rail part that read from input file.
 * @param details pointer to array of part details.
 * @param joints array of rail joints.
 * @return true if it valid, else, false.
 */
bool checkPartDetails(char part[], char **details, char joints[])
{
    char *cutPtr = strtok(part, "\n");   // cut the newline char
    char *ptr = strtok(cutPtr, ",");
    int idx = 0;
    while (ptr != NULL)
    {
        if (! helperCheckPartDetails(ptr, idx, joints))
        {
            return false;
        }
        // if part detail is valid then add it to the details array.
        // details[0] -> start, details[1] -> end,
        // details[2] -> length, details[3] -> price.
        // These dynamic allocated arrays will be freed later in addPartDetail() function.
        details[idx] = (char *)malloc(strlen(ptr) * sizeof(char));
        strncpy(details[idx], ptr, strlen(ptr));
        ptr = strtok(NULL, ",");
        idx++;
    }
    return true;
}

/**
 * @brief helper function to check part details validity.
 * @param detail detail about the part
 * @param idx index shows what each detail stands for
 * @param joints
 * @return
 */
bool helperCheckPartDetails(char detail[], int idx, char joints[])
{
    // what idx represents : [0] -> start, [1] -> end,
    if (idx == 0 || idx == 1)
    {
        // if start and end joints in joints array, and they are just one char.
        if ((strstr(joints, &detail[0]) == NULL) || strlen(&detail[0]) > 1)
        {
            return false;
        }
    }
    // what idx represents : [2] -> length, [3] -> price.
    else if (idx == 2 || idx == 3)
    {
        if (checkInteger(detail) <= 0)
        {
            return false;
        }
    }
    return true;
}

/**
 * @brief adds part with its details to parts collection.
 * @param details start, end, length, price of the part.
 * @param railInfo a struct contains all rail info
 * @param capacity capacity of array.
 * @param line the line the detail shows at in the input file.
 * @return true if adding part succeeded, false if not.
 */
bool addPartDetail(char **details, AlgIn *railInfo, int *capacity, int line)
{
    for (int j = 0; j < *capacity; j++)  // check if part already existed
    {
        if (partExists(details, railInfo->partsCollection[j]))
        {
            // free the dynamic sub-arrays of details, that previously
            // allocated in checkPartDetails() function
            for (int i = 0; i < NUM_PART_DETAILS; i++)
            {
                free(details[i]);
                details[i] = NULL;
            }
            return false;
        }
    }
    int numParts = line - 3;   // first 3 lines in input file are not for part details
    if (numParts == *capacity)   // resize parts collection array as needed
    {
        *capacity += INITIAL_ALLOC;
        railInfo->partsCollection = (railPart *)realloc(railInfo->partsCollection,
                                                        *capacity * sizeof(railPart));
    }

    sscanf(details[0], "%c", &railInfo->partsCollection[numParts - 1].start);
    sscanf(details[1], "%c", &railInfo->partsCollection[numParts - 1].end);

    char *lenPtr, *pricePtr;
    railInfo->partsCollection[numParts - 1].length = (int) strtol(details[2], &lenPtr, 10);
    railInfo->partsCollection[numParts - 1].price = (int) strtol(details[3], &pricePtr, 10);

    // free the dynamic sub-arrays of details, that previously
    // allocated in checkPartDetails() function
    for (int i = 0; i < NUM_PART_DETAILS; i++)
    {
        free(details[i]);
        details[i] = NULL;
    }

    return true;
}

/**
 * @brief compile all input-check functions into one function
 * @param file input file
 * @param railInfo a pointer to struct contains all rail info.
 * @return number of checked line in input file if its a valid line, else returns 0.
 */
int checkDetails(FILE *file, AlgIn *railInfo)
{
    char input[MAX_ROW] = {0};
    char *partDetails[NUM_PART_DETAILS] = {NULL};
    int capacity = INITIAL_ALLOC;

    // this dynamic array will be freed later in freeAll() function.
    railInfo -> partsCollection = (railPart *)calloc(capacity, sizeof(railPart));
    if (railInfo -> partsCollection == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int line = 1;
    while (fgets(input, sizeof(input), file) != NULL)
    {
        if (line == 1)   // check input at first line (rail length)
        {
            railInfo -> railLen = checkInteger(input);
            if (railInfo -> railLen < 0)
            {
                return line;
            }
        }

        else if (line == 2)   // check input at second line (number of joints)
        {
            railInfo -> numJoints = checkInteger(input);
            if (railInfo -> numJoints <= 0)
            {
                return line;
            }
        }

        else if (line == 3)   // check input at third line (kinds of joints)
        {
            if (! checkJoints(input, railInfo -> numJoints))
            {
                return line;
            }
            railInfo -> kindsJoints = jointsArray(input, railInfo -> numJoints);
        }

        else   // check input from forth line (parts details)
        {
            if (! checkPartDetails(input, partDetails, railInfo -> kindsJoints))
            {
                return line;
            }
            if (! addPartDetail(partDetails, railInfo, &capacity, line))
            {
                // part is already exists, so number of rail parts in parts collection
                // not changed, so this line not added to num of parts.
                line--;
            }
            railInfo -> numParts = line - 3;
        }
        line++;
    }
    return 0;
}

/**
 * @brief return index of joint in the kindsJoints array.
 * @param joint a joint of rail parts
 * @param railInfo a struct contains all rail info.
 * @return index of joint, or NO_INDEX (= -1) if it is not in array.
 */
int jointIDX(char joint, AlgIn railInfo)
{
    for (int i = 0; i < railInfo.numJoints; i++)
    {
        if (joint == railInfo.kindsJoints[i])
        {
            return i;
        }
    }
    return NO_INDEX;
}

/**
 * @brief calculates the min cost of rail of length (row) ended with a given joint
 * @param joint a joint of rail parts
 * @param row row in cost table, represents length of rail.
 * @param table table of costs of optimal paths to build the rail.
 * @param railInfo a struct contains all rail info.
 * @return min cost of rail of length (row) ended with a given joint,
 *         or NOT_POSSIBLE (= INT_MAX) if it is not possible to build such a rail.
 */
int min(char joint, int row, const int table[], AlgIn railInfo)
{
    // By dynamic programing not by recursion, we calculate the min cost in
    // a row depending on previous rows only, by this formula:
    // T[r][c] = P(i) + T[r - D(i)][idx(S(i))]

    unsigned long minCost = NOT_POSSIBLE;
    for (int i = 0; i < railInfo.numParts; i++)
    {
        if (railInfo.partsCollection[i].end == joint &&
            railInfo.partsCollection[i].length <= row)
        {
            // get index of start joint of this part
            int colIdx = jointIDX(railInfo.partsCollection[i].start, railInfo);
            unsigned long cost = railInfo.partsCollection[i].price +
                    table[(row - railInfo.partsCollection[i].length) * railInfo.numJoints + colIdx];

            if (cost < minCost)
            {
                minCost = cost;
            }
        }
    }
    return (int) minCost;
}

/**
 * @brief builds the table of optimal costs to build the rail.
 * @param rows rows of table (= rail length + 1)
 * @param columns columns of table (= number of joints)
 * @param railInfo a struct contains all rail info.
 * @return the costs table
 */
int *tableBuilder(int rows, int columns, AlgIn railInfo)
{
    // this dynamic array will be freed later in freeAll() function.
    int *table = (int *)malloc(rows * columns * sizeof(int));
    if (table == NULL)
    {
        exit(EXIT_FAILURE);
    }

    for (int r = 0; r < rows; r++)
    {
        for (int c = 0; c < columns; c++)
        {
            if (r == 0)
            {
                table[r * columns + c] = 0;
            }
            else
            {
                table[r * columns + c] = min(railInfo.kindsJoints[c], r, table, railInfo);
            }
        }
    }
    return table;
}

/**
 * @brief calculate min cost to build the rail given the details in inputfile
 * @param table table of costs of optimal paths to build the rail.
 * @param railInfo a struct contains all rail info.
 * @return minimal cost if there is, or CANNOT_BUILD (= -1) if not.
 */
int minTotalCost(const int *table, AlgIn railInfo)
{
    int minTotal = table[railInfo.railLen * railInfo.numJoints];
    for (int i = 1; i < railInfo.numJoints; i++)
    {
        int curCost = table[railInfo.railLen * railInfo.numJoints + i];
        if (curCost < minTotal)
        {
            minTotal = curCost;
        }
    }
    if (minTotal == NOT_POSSIBLE)
    {
        return CANNOT_BUILD;
    }
    return minTotal;
}


/**
 * @brief EXTRA function to PRINT the table on the screen.
 *        <(uncomment its call in the main function)>
 * @param table table of costs of optimal paths to build the rail.
 * @param railInfo a struct contains all rail info.
 */
void printTable(const int *table, AlgIn railInfo)
{
    int rows = railInfo.railLen + 1;
    int columns = railInfo.numJoints;
    printf("   ");
    for (int c = 0; c < columns; c++)
    {
        printf("\t[ %c ]\t", railInfo.kindsJoints[c]);
    }
    printf("\n");

    for (int c = 0; c < columns; c++)
    {
        printf("\t-----\t");
    }
    printf("\n");

    for (int r = 0; r < rows; r++)
    {
        printf("(%d)", r);
        for (int c = 0; c < columns; c++)
        {
            if (table[r * columns + c] == NOT_POSSIBLE)
            {
                printf("\tX\t|");
            }
            else
            {
                printf("\t%d\t|", table[r * columns + c]);
            }
        }
        printf("\n");
    }
}

/**
 * @brief frees all remaining dynamic allocated arrays.
 * @param table pointer to table of costs of optimal paths to build the rail.
 * @param railInfo pointer to the struct contains all rail info.
 */
void freeAll(int **table, AlgIn *railInfo)
{
    // allocated in tableBuilder() function
    free(*table);
    *table = NULL;

    // allocated in jointsArray() function
    free(railInfo -> kindsJoints);
    railInfo -> kindsJoints = NULL;

    // allocated in checkDetails() function
    free(railInfo -> partsCollection);
    railInfo -> partsCollection = NULL;
}

/**
 * @brief The main function.
 *        Opens hte input file, and closes it at the end,
 *        checks validity of input file,
 *        calculates the min cost.
 *        prints the cost table.
 * @return EXIT_SUCCESS, to tell the system the execution ended without errors,
 *         otherwise, EXIT_FAILURE.
 */
int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        outputMessage(ARGC_ERROR, NO_NUM);
        return EXIT_FAILURE;
    }

    FILE *inputFile = fopen(argv[1], "r");

    if (! validFile(inputFile))
    {
        fclose(inputFile);
        return EXIT_FAILURE;
    }

    AlgIn inputDetails = {0};
    int line = checkDetails(inputFile, &inputDetails);
    if (line)
    {
        outputMessage(INVALID_INPUT_ERROR, line);
        return EXIT_FAILURE;
    }

    int *costsTable = tableBuilder(inputDetails.railLen + 1, inputDetails.numJoints, inputDetails);

    int minPrice = minTotalCost(costsTable, inputDetails);
    outputMessage(MIN_PRICE_RES, minPrice);

//    printTable(costsTable, inputDetails);   // uncomment to print the table   <==(*)

    freeAll(&costsTable, &inputDetails);
    fclose(inputFile);

    return EXIT_SUCCESS;
}
```