

Contents

1	Basic Test Results	2
2	RailWayPlanner.c	4

1 Basic Test Results

```
1 Running...
2 Opening tar file
3 OK
4 Tar extracted O.K.
5 Checking files...
6 OK
7 Making sure files are not empty...
8 OK
9 Compilation check...
10 Compiling...
11 OK
12 Compilation seems OK! Check if you got warnings!
13
14 =====
15 Public test cases
16 =====
17
18 =====
19 Test #1
20 Running RailWayPlanner
21 OK
22 Running diff
23 OK
24 Test 1 passed.
25 =====
26
27 =====
28 Test #2
29 Running RailWayPlanner
30 OK
31 Running diff
32 OK
33 Test 2 passed.
34 =====
35
36 =====
37 Test #3
38 Running RailWayPlanner
39 OK
40 Running diff
41 OK
42 Test 3 passed.
43 =====
44
45 =====
46 Test #4
47 Running RailWayPlanner
48 OK
49 Running diff
50 OK
51 Test 4 passed.
52 =====
53
54 =====
55 Test #5
56 Running RailWayPlanner
57 OK
58 Running diff
59 OK
```

```

60 Test 5 passed.
61 =====
62
63 =====
64 Test #6
65 Running RailWayPlanner
66 OK
67 Running diff
68 OK
69 Test 6 passed.
70 =====
71
72 =====
73 Test #7
74 Running RailWayPlanner
75 OK
76 Running diff
77 OK
78 Test 7 passed.
79 =====
80
81 =====
82 Test #8
83 Running RailWayPlanner
84 OK
85 Running diff
86 OK
87 Test 8 passed.
88 =====
89
90 =====
91 Test #9
92 Running RailWayPlanner
93 OK
94 Running diff
95 OK
96 Test 9 passed.
97 =====
98
99 =====
100 Test #10
101 Running RailWayPlanner
102 OK
103 Running diff
104 OK
105 Test 10 passed.
106 =====
107
108 *****
109 *                                     *
110 *   presubmission script passed   *
111 *       10/10 tests passed       *
112 *                                     *
113 *****
114
115 =====
116 = Checking coding style =
117 =====
118 ** Total Violated Rules      : 0
119 ** Total Errors Occurs      : 0
120 ** Total Violated Files Count: 0

```

2 RailWayPlanner.c

```
1  /**
2   * @file RailWayPlanner.c
3   * @author Muaz Abdeen <muaz.abdeen@mail.huji.ac.il>
4   * @ID 300575297
5   * @date 7 May 2020
6   *
7   * @brief Program that receives from user a file contains rail details:
8   *         - Rail length
9   *         - Number of rail joints
10  *         - Kinds of joints
11  *         - Parts of rail <start joint, end joint, length, price>
12  *         and given a valid input it calculates the minimal cost to build a rail of the
13  *         given length from the given parts.
14  *
15  * @section DESCRIPTION
16  * Program that calculates the minimal cost of the rail.
17  * Input : Rail's info: <length>,<number of joints>,<kinds of joints>,<building parts>
18  * Process: given a valid input it calculates the minimal cost to build a rail of the
19  *         given length from the given parts.
20  * Output : > If the input is not valid - print informative message.
21  *         > Else if the input is valid - print the minimal cost.
22  */
23
24 // ----- includes -----
25
26 #include <stdio.h>
27 #include <stdlib.h>
28 #include <string.h>
29 #include <limits.h>
30 #include <ctype.h>
31 #include <regex.h>
32 #include <stdbool.h>
33
34 // ----- macros & constants -----
35
36 #define MAX_ROW 1024
37 #define NUM_PART_DETAILS 4
38 #define INITIAL_ALLOC 10
39 #define NO_NUM -5
40 #define NOT_POSSIBLE INT_MAX
41 #define NO_INDEX -1
42 #define CANNOT_BUILD -1
43 #define OUTPUT_FILE "./railway_planner_output.txt"
44 #define ARGV_ERROR "Usage: RailWayPlanner <InputFile>"
45 #define NO_FILE_ERROR "File doesn't exists."
46 #define EMPTY_FILE_ERROR "File is empty."
47 #define INVALID_INPUT_ERROR "Invalid input in line: %d."
48 #define MIN_PRICE_RES "The minimal price is: %d"
49
50 // ----- functions & structs -----
51
52 typedef struct railPart railPart;
53 typedef struct algorithmInput AlgIn;
54
55 void outputMessage(char *arr, int num);
56 bool emptyFile(FILE *file);
57 bool validFile(FILE *filename);
58 bool checkDigit(char arr[]);
59 int checkInteger(char arr[]);
```

```

60 bool checkJoints(char arr[], int numJoints);
61 char *jointsArray(char arr[], int numJoints);
62 bool helperCheckPartDetails(char detail[], int idx, char joints[]);
63 bool addPartDetail(char **details, AlgIn *railInfo, int *capacity, int line);
64 int checkDetails(FILE *file, AlgIn *railInfo);
65 int jointIDX(char joint, AlgIn railInfo);
66 int min(char joint, int row, const int table[], AlgIn railInfo);
67 int *tableBuilder(int rows, int columns, AlgIn railInfo);
68 int minTotalCost(const int *table, AlgIn railInfo);
69 void printTable(const int *table, AlgIn railInfo); // Extra function to display the table
70 void freeAll(int **table, AlgIn *railInfo);
71
72 // -----
73
74 /**
75  * @brief A structure to represent rail part details.
76  */
77 struct railPart
78 {
79     char start, end; /**< starting and ending joints */
80     int length, price;
81 };
82
83
84 /**
85  * @brief A structure to represent the rail info which received from input file.
86  */
87 struct algorithmInput
88 {
89     int railLen, numJoints, numParts;
90     char *kindsJoints; /**< different kinds of joints */
91     railPart *partsCollection; /**< collection of all parts to be used in rail construction */
92 };
93
94
95 /**
96  * @brief A function to print a suitable message to output file.
97  * @param arr C string represents the message
98  * @param num number of line (in case it exists)
99  */
100 void outputMessage(char *arr, int num)
101 {
102     FILE *outputFile = fopen(OUTPUT_FILE, "w");
103     if (outputFile == NULL)
104     {
105         exit(EXIT_FAILURE);
106     }
107     if (num == NO_NUM) // the message doesn't contains a line number
108     {
109         fprintf(outputFile, "%s", arr);
110     }
111     else
112     {
113         fprintf(outputFile, arr, num);
114     }
115     fclose(outputFile);
116 }
117
118
119 /**
120  * @brief A function to check if input file is empty.
121  * @param file the input file.
122  * @return true if empty, else, false.
123  */
124 bool emptyFile(FILE *file)
125 {
126     fseek(file, 0, SEEK_END);
127     if (ftell(file) == 0)

```

```

128     {
129         return true;
130     }
131     rewind(file);    // return the pointer to the start
132     return false;
133 }
134
135 /**
136  * @brief A function to check if input file is exists and not empty.
137  * @param file input file.
138  * @return true if valid, else, false.
139  */
140 bool validFile(FILE *file)
141 {
142     if (file == NULL)    // fopen in main returns NULL
143     {
144         outputMessage(NO_FILE_ERROR, NO_NUM);
145         return false;
146     }
147     if (emptyFile(file))
148     {
149         outputMessage(EMPTY_FILE_ERROR, NO_NUM);
150         return false;
151     }
152     return true;
153 }
154
155 /**
156  * @brief A function to check if chars in string are numbers only.
157  * @param arr string supposed to represent a number.
158  * @return true if it contains only numbers, else, false.
159  */
160 bool checkDigit(char arr[])
161 {
162     char *cutInput = (char*)malloc((int) strlen(arr) * sizeof(char) + 1);
163     sscanf(arr, "%[^\n]", cutInput);    // cut the newline char
164     for (int i = 0; i < (int) strlen(cutInput); i++)
165     {
166         if (!isdigit(arr[i]))
167         {
168             return false;
169         }
170     }
171     free(cutInput);
172     // cutInput = NULL;
173     return true;
174 }
175
176 /**
177  * @brief A function to check and convert string to integer.
178  * @param arr string supposed to represent a number.
179  * @return the integer that represented in the string, ot -1 if it fails.
180  */
181 int checkInteger(char arr[])
182 {
183     if (!checkDigit(arr))
184     {
185         return -1;
186     }
187     char *ptr;
188     int num = (int) strtol(arr, &ptr, 10);
189     return num;
190 }
191
192 /**
193  * @brief checks validity of joints in input file.
194  * @param arr string contains all joints.
195  * @param numJoints number of joints.

```

```

196  * @return false if any of the joints is more than one char, or their number exceeds
197  *       the provided number in input file. else returns true.
198  */
199  bool checkJoints(char arr[], int numJoints)
200  {
201      // check if # of joint symbols exceeds # of joint kinds
202      int count = 0;
203      for (int i = 0; i < (int) strlen(arr) - 1; i++)
204      {
205          if (arr[i] == ',')
206          {
207              count++;
208          }
209      }
210
211      if (count != (numJoints - 1))
212      {
213          return false;
214      }
215
216      // check if there is joint symbols with more than one char
217      int jointSymbols = (int) strlen(arr) - count - 1;
218      if (jointSymbols != numJoints)
219      {
220          return false;
221      }
222      return true;
223  }
224
225  /**
226   * @brief builds a char array contains the joints.
227   * @param arr string contains all joints separated by comma.
228   * @param numJoints number of joints provided in input file.
229   * @return the rail joints array.
230   */
231  char *jointsArray(char arr[], int numJoints)
232  {
233      // this dynamic array will be freed later in freeAll() function.
234      char *railJoints = (char *)malloc(numJoints * sizeof(char) + 1);
235      if (railJoints == NULL)
236      {
237          exit(EXIT_FAILURE);
238      }
239      char *ptr = strtok(arr, ",");
240      for (int i = 0; i < numJoints; i++)
241      {
242          if (ptr != NULL)
243          {
244              railJoints[i] = *ptr;
245              ptr = strtok(NULL, ",");
246          }
247      }
248      return railJoints;
249  }
250
251  /**
252   * @brief checks if a rail part is already existed in parts collection.
253   * @param newPart a new part to be added to the collection.
254   * @param existedPart an already existed part.
255   * @return true if it is already existed, else, false.
256   */
257  bool partExists(char *newPart[], railPart existedPart)
258  {
259      char *lenPtr, *pricePtr;
260      int partLen = (int) strtol(newPart[2], &lenPtr, 10);
261      int partPrice = (int) strtol(newPart[3], &pricePtr, 10);
262
263      if ((newPart[0][0] == existedPart.start) && (newPart[1][0] == existedPart.end) &&

```

```

264         (partLen == existedPart.length) && (partPrice == existedPart.price))
265     {
266         return true;
267     }
268     return false;
269 }
270
271 /**
272  * @brief checks for every part if its details are valid
273  * @param part rail part that read from input file.
274  * @param details pointer to array of part details.
275  * @param joints array of rail joints.
276  * @return true if it valid, else, false.
277  */
278 bool checkPartDetails(char part[], char **details, char joints[])
279 {
280     char *cutPtr = strtok(part, "\n");    // cut the newline char
281     char *ptr = strtok(cutPtr, ",");
282     int idx = 0;
283     while (ptr != NULL)
284     {
285         if (! helperCheckPartDetails(ptr, idx, joints))
286         {
287             return false;
288         }
289         // if part detail is valid then add it to the details array.
290         // details[0] -> start, details[1] -> end,
291         // details[2] -> length, details[3] -> price.
292         // These dynamic allocated arrays will be freed later in addPartDetail() function.
293         details[idx] = (char *)malloc(strlen(ptr) * sizeof(char));
294         strncpy(details[idx], ptr, strlen(ptr));
295         ptr = strtok(NULL, ",");
296         idx++;
297     }
298     return true;
299 }
300
301 /**
302  * @brief helper function to check part details validity.
303  * @param detail detail about the part
304  * @param idx index shows what each detail stands for
305  * @param joints
306  * @return
307  */
308 bool helperCheckPartDetails(char detail[], int idx, char joints[])
309 {
310     // what idx represents : [0] -> start, [1] -> end,
311     if (idx == 0 || idx == 1)
312     {
313         // if start and end joints in joints array, and they are just one char.
314         if ((strstr(joints, &detail[0]) == NULL) || strlen(&detail[0]) > 1)
315         {
316             return false;
317         }
318     }
319     // what idx represents : [2] -> length, [3] -> price.
320     else if (idx == 2 || idx == 3)
321     {
322         if (checkInteger(detail) <= 0)
323         {
324             return false;
325         }
326     }
327     return true;
328 }
329
330 /**
331  * @brief adds part with its details to parts collection.

```



```

332  * @param details start, end, length, price of the part.
333  * @param railInfo a struct contains all rail info
334  * @param capacity capacity of array.
335  * @param line the line the detail shows at in the input file.
336  * @return true if adding part succeeded, false if not.
337  */
338  bool addPartDetail(char **details, AlgIn *railInfo, int *capacity, int line)
339  {
340      for (int j = 0; j < *capacity; j++) // check if part already existed
341      {
342          if (partExists(details, railInfo->partsCollection[j]))
343          {
344              // free the dynamic sub-arrays of details, that previously
345              // allocated in checkPartDetails() function
346              for (int i = 0; i < NUM_PART_DETAILS; i++)
347              {
348                  free(details[i]);
349                  details[i] = NULL;
350              }
351              return false;
352          }
353      }
354      int numParts = line - 3; // first 3 lines in input file are not for part details
355      if (numParts == *capacity) // resize parts collection array as needed
356      {
357          *capacity += INITIAL_ALLOC;
358          railInfo->partsCollection = (railPart *)realloc(railInfo->partsCollection,
359                                                         *capacity * sizeof(railPart));
360      }
361
362      sscanf(details[0], "%c", &railInfo->partsCollection[numParts - 1].start);
363      sscanf(details[1], "%c", &railInfo->partsCollection[numParts - 1].end);
364
365      char *lenPtr, *pricePtr;
366      railInfo->partsCollection[numParts - 1].length = (int) strtol(details[2], &lenPtr, 10);
367      railInfo->partsCollection[numParts - 1].price = (int) strtol(details[3], &pricePtr, 10);
368
369      // free the dynamic sub-arrays of details, that previously
370      // allocated in checkPartDetails() function
371      for (int i = 0; i < NUM_PART_DETAILS; i++)
372      {
373          free(details[i]);
374          details[i] = NULL;
375      }
376
377      return true;
378  }
379
380  /**
381   * @brief compile all input-check functions into one function
382   * @param file input file
383   * @param railInfo a pointer to struct contains all rail info.
384   * @return number of checked line in input file if its a valid line, else returns 0.
385   */
386  int checkDetails(FILE *file, AlgIn *railInfo)
387  {
388      char input[MAX_ROW] = {0};
389      char *partDetails[NUM_PART_DETAILS] = {NULL};
390      int capacity = INITIAL_ALLOC;
391
392      // this dynamic array will be freed later in freeAll() function.
393      railInfo-> partsCollection = (railPart *)calloc(capacity, sizeof(railPart));
394      if (railInfo-> partsCollection == NULL)
395      {
396          exit(EXIT_FAILURE);
397      }
398
399      int line = 1;

```

```

400 while (fgets(input, sizeof(input), file) != NULL)
401 {
402     if (line == 1) // check input at first line (rail length)
403     {
404         railInfo -> railLen = checkInteger(input);
405         if (railInfo -> railLen < 0)
406         {
407             return line;
408         }
409     }
410
411     else if (line == 2) // check input at second line (number of joints)
412     {
413         railInfo -> numJoints = checkInteger(input);
414         if (railInfo -> numJoints <= 0)
415         {
416             return line;
417         }
418     }
419
420     else if (line == 3) // check input at third line (kinds of joints)
421     {
422         if (! checkJoints(input, railInfo -> numJoints))
423         {
424             return line;
425         }
426         railInfo -> kindsJoints = jointsArray(input, railInfo -> numJoints);
427     }
428
429     else // check input from forth line (parts details)
430     {
431         if (! checkPartDetails(input, partDetails, railInfo -> kindsJoints))
432         {
433             return line;
434         }
435         if (! addPartDetail(partDetails, railInfo, &capacity, line))
436         {
437             // part is already exists, so number of rail parts in parts collection
438             // not changed, so this line not added to num of parts.
439             line--;
440         }
441         railInfo -> numParts = line - 3;
442     }
443     line++;
444 }
445 return 0;
446 }
447
448 /**
449  * @brief return index of joint in the kindsJoints array.
450  * @param joint a joint of rail parts
451  * @param railInfo a struct contains all rail info.
452  * @return index of joint, or NO_INDEX (= -1) if it is not in array.
453  */
454 int jointIDX(char joint, AlgIn railInfo)
455 {
456     for (int i = 0; i < railInfo.numJoints; i++)
457     {
458         if (joint == railInfo.kindsJoints[i])
459         {
460             return i;
461         }
462     }
463     return NO_INDEX;
464 }
465
466 /**
467  * @brief calculates the min cost of rail of length (row) ended with a given joint

```

```

468  * @param joint a joint of rail parts
469  * @param row row in cost table, represents length of rail.
470  * @param table table of costs of optimal paths to build the rail.
471  * @param railInfo a struct contains all rail info.
472  * @return min cost of rail of length (row) ended with a given joint,
473  *         or NOT_POSSIBLE (= INT_MAX) if it is not possible to build such a rail.
474  */
475  int min(char joint, int row, const int table[], AlgIn railInfo)
476  {
477      // By dynamic programming not by recursion, we calculate the min cost in
478      // a row depending on previous rows only, by this formula:
479      //  $T[r][c] = P(i) + T[r - D(i)][idx(S(i))]$ 
480
481      unsigned long minCost = NOT_POSSIBLE;
482      for (int i = 0; i < railInfo.numParts; i++)
483      {
484          if (railInfo.partsCollection[i].end == joint &&
485              railInfo.partsCollection[i].length <= row)
486          {
487              // get index of start joint of this part
488              int colIdx = jointIDX(railInfo.partsCollection[i].start, railInfo);
489              unsigned long cost = railInfo.partsCollection[i].price +
490                  table[(row - railInfo.partsCollection[i].length) * railInfo.numJoints + colIdx];
491
492              if (cost < minCost)
493              {
494                  minCost = cost;
495              }
496          }
497      }
498      return (int) minCost;
499  }
500
501  /**
502   * @brief builds the table of optimal costs to build the rail.
503   * @param rows rows of table (= rail length + 1)
504   * @param columns columns of table (= number of joints)
505   * @param railInfo a struct contains all rail info.
506   * @return the costs table
507   */
508  int *tableBuilder(int rows, int columns, AlgIn railInfo)
509  {
510      // this dynamic array will be freed later in freeAll() function.
511      int *table = (int *)malloc(rows * columns * sizeof(int));
512      if (table == NULL)
513      {
514          exit(EXIT_FAILURE);
515      }
516
517      for (int r = 0; r < rows; r++)
518      {
519          for (int c = 0; c < columns; c++)
520          {
521              if (r == 0)
522              {
523                  table[r * columns + c] = 0;
524              }
525              else
526              {
527                  table[r * columns + c] = min(railInfo.kindsJoints[c], r, table, railInfo);
528              }
529          }
530      }
531      return table;
532  }
533
534  /**
535   * @brief calculate min cost to build the rail given the details in inputfile

```

```

536  * @param table table of costs of optimal paths to build the rail.
537  * @param railInfo a struct contains all rail info.
538  * @return minimal cost if there is, or CANNOT_BUILD (= -1) if not.
539  */
540  int minTotalCost(const int *table, AlgIn railInfo)
541  {
542      int minTotal = table[railInfo.railLen * railInfo.numJoints];
543      for (int i = 1; i < railInfo.numJoints; i++)
544      {
545          int curCost = table[railInfo.railLen * railInfo.numJoints + i];
546          if (curCost < minTotal)
547          {
548              minTotal = curCost;
549          }
550      }
551      if (minTotal == NOT_POSSIBLE)
552      {
553          return CANNOT_BUILD;
554      }
555      return minTotal;
556  }
557
558  /**
559  * @brief EXTRA function to PRINT the table on the screen.
560  *      <(uncomment its call in the main function)>
561  * @param table table of costs of optimal paths to build the rail.
562  * @param railInfo a struct contains all rail info.
563  */
564  void printTable(const int *table, AlgIn railInfo)
565  {
566      int rows = railInfo.railLen + 1;
567      int columns = railInfo.numJoints;
568      printf(" ");
569      for (int c = 0; c < columns; c++)
570      {
571          printf("\t[ %c ]\t", railInfo.kindsJoints[c]);
572      }
573      printf("\n");
574
575      for (int c = 0; c < columns; c++)
576      {
577          printf("\t----\t");
578      }
579      printf("\n");
580
581      for (int r = 0; r < rows; r++)
582      {
583          printf("(%d)", r);
584          for (int c = 0; c < columns; c++)
585          {
586              if (table[r * columns + c] == NOT_POSSIBLE)
587              {
588                  printf("\tX\t");
589              }
590              else
591              {
592                  printf("\t%d\t", table[r * columns + c]);
593              }
594          }
595          printf("\n");
596      }
597  }
598
599  /**
600  * @brief frees all remaining dynamic allocated arrays.
601  * @param table pointer to table of costs of optimal paths to build the rail.
602  * @param railInfo pointer to the struct contains all rail info.

```

```

604  */
605  void freeAll(int **table, AlgIn *railInfo)
606  {
607      // allocated in tableBuilder() function
608      free(*table);
609      *table = NULL;
610
611      // allocated in jointsArray() function
612      free(railInfo -> kindsJoints);
613      railInfo -> kindsJoints = NULL;
614
615      // allocated in checkDetails() function
616      free(railInfo -> partsCollection);
617      railInfo -> partsCollection = NULL;
618  }
619
620  /**
621   * @brief The main function.
622   *      Opens hte input file, and closes it at the end,
623   *      checks validity of input file,
624   *      calculates the min cost.
625   *      prints the cost table.
626   * @return EXIT_SUCCESS, to tell the system the execution ended without errors,
627   *      otherwise, EXIT_FAILURE.
628   */
629  int main(int argc, char *argv[])
630  {
631      if (argc != 2)
632      {
633          outputMessage(ARGC_ERROR, NO_NUM);
634          return EXIT_FAILURE;
635      }
636
637      FILE *inputFile = fopen(argv[1], "r");
638
639      if (! validFile(inputFile))
640      {
641          fclose(inputFile);
642          return EXIT_FAILURE;
643      }
644
645      AlgIn inputDetails = {0};
646      int line = checkDetails(inputFile, &inputDetails);
647      if (line)
648      {
649          outputMessage(INVALID_INPUT_ERROR, line);
650          return EXIT_FAILURE;
651      }
652
653      int *costsTable = tableBuilder(inputDetails.railLen + 1, inputDetails.numJoints, inputDetails);
654
655      int minPrice = minTotalCost(costsTable, inputDetails);
656      outputMessage(MIN_PRICE_RES, minPrice);
657
658      // printTable(costsTable, inputDetails); // uncomment to print the table <==(*)
659
660      freeAll(&costsTable, &inputDetails);
661      fclose(inputFile);
662
663      return EXIT_SUCCESS;
664  }

```