```c
/* File:     tsp_rec.c
 * Purpose:  Use recursive depth-first search to solve an instance of the
 *           travelling salesman problem.
 *
 * Compile:  gcc -g -Wall -o tsp_rec tsp_rec.c
 *           Needs timer.h
 * Usage:    tsp_rec <matrix_file>
 *
 * Input:    From a user-specified file, the number of cities
 *           followed by the costs of travelling between the
 *           cities organized as a matrix:  the cost of
 *           travelling from city i to city j is the ij entry.
 *           Costs are nonnegative ints.  Diagonal entries are 0.
 * Output:   The best tour found by the program and the cost
 *           of the tour.
 *
 * Notes:
 * 1.  Costs and cities are non-negative ints.
 * 2.  Program assumes the cost of travelling from a city to
 *     itself is zero, and the cost of travelling from one
 *     city to another city is positive.
 * 3.  Note that costs may not be symmetric:  the cost of travelling
 *     from A to B, may, in general, be different from the cost
 *     of travelling from B to A.
 * 4.  Salesperson's home town is 0.
 * 5.  The digraph is stored as an adjacency matrix, which is
 *     a one-dimensional array:  digraph[i][j] is computed as
 *     digraph[i*n + j]
 * 6.  Debug option prints verbose output
 *
 * IPP:  Section 6.2.1 (pp. 302 and ff.)
 */
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"

const int INFINITY = 1000000;
const int NO_CITY = -1;
const int FALSE = 0;
const int TRUE = 1;

typedef int city_t;
typedef int cost_t;
typedef struct {
   city_t* cities; /* Cities in partial tour           */
   int count;      /* Number of cities in partial tour */
   cost_t cost;    /* Cost of partial tour             */
} tour_struct;
typedef tour_struct* tour_t;
#define City_count(tour) (tour->count)
```

```c
#define Tour_cost(tour) (tour->cost)
#define Last_city(tour) (tour->cities[(tour->count)-1])
#define Tour_city(tour,i) (tour->cities[(i)])

/* Global Vars:  Except for best_tour, all are constant after initializatio
int n;  /* Number of cities in the problem */
cost_t* digraph;
city_t home_town = 0;
#ifdef DEBUG
long call_count = 0;
#endif
tour_t best_tour;
#define Cost(city1, city2) (digraph[city1*n + city2])

void Usage(char* prog_name);
void Read_digraph(FILE* digraph_file);
void Print_digraph(void);

void Depth_first_search(tour_t tour);
void Print_tour(tour_t tour, char* title);
int  Best_tour(tour_t tour);
void Update_best_tour(tour_t tour);
void Copy_tour(tour_t tour1, tour_t tour2);
void Add_city(tour_t tour, city_t);
void Remove_last_city(tour_t tour);
int  Feasible(tour_t tour, city_t city);
int  Visited(tour_t tour, city_t city);
void Init_tour(tour_t tour, cost_t cost);

/*------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   FILE* digraph_file;
   tour_t tour;
   double start, finish;

   if (argc != 2) Usage(argv[0]);
   digraph_file = fopen(argv[1], "r");
   if (digraph_file == NULL) {
      fprintf(stderr, "Can't open %s\n", argv[1]);
      Usage(argv[0]);
   }
   Read_digraph(digraph_file);
   fclose(digraph_file);
#  ifdef DEBUG
   Print_digraph();
#  endif

   best_tour = malloc(sizeof(tour_struct));
   Init_tour(best_tour, INFINITY);
   tour = malloc(sizeof(tour_struct));
   Init_tour(tour, 0);

   GET_TIME(start);
   Depth_first_search(tour);
   GET_TIME(finish);

   Print_tour(best_tour, "Best tour");
   printf("Cost = %d\n", best_tour->cost);
   printf("Elapsed time = %e seconds\n", finish-start);

   free(best_tour->cities);
   free(best_tour);
   free(tour->cities);
   free(tour);
   free(digraph);
   return 0;
}  /* main */

/*------------------------------------------------------------------
 * Function:  Init_tour
 * Purpose:   Allocate storage for the cities on the tour, and
 *            initialize the data members
 * In args:
 *    cost:   initial cost of tour
 * Global in:
 *    n:      number of cities in TSP
 * Out arg:
 *    tour
 */
void Init_tour(tour_t tour, cost_t cost) {
   int i;

   tour->cities = malloc((n+1)*sizeof(city_t));
   tour->cities[0] = 0;
   for (i = 1; i <= n; i++) {
      tour->cities[i] = NO_CITY;
   }
   tour->cost = cost;
   tour->count = 1;
}  /* Init_tour */


/*------------------------------------------------------------------
 * Function:  Usage
 * Purpose:   Inform user how to start program and exit
 * In arg:    prog_name
 */
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <digraph file>\n", prog_name);
   exit(0);
```

```c
}  /* Usage */

/*------------------------------------------------------------------
 * Function:  Read_digraph
 * Purpose:   Read in the number of cities and the digraph of costs
 * In arg:    digraph_file
 * Globals out:
 *    n:        the number of cities
 *    digraph:  the matrix file
 */
void Read_digraph(FILE* digraph_file) {
   int i, j;

   fscanf(digraph_file, "%d", &n);
   if (n <= 0) {
      fprintf(stderr, "Number of vertices in digraph must be positive\n");
      exit(-1);
   }
   digraph = malloc(n*n*sizeof(cost_t));

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         fscanf(digraph_file, "%d", &digraph[i*n + j]);
         if (i == j && digraph[i*n + j] != 0) {
            fprintf(stderr, "Diagonal entries must be zero\n");
            exit(-1);
         } else if (i != j && digraph[i*n + j] <= 0) {
            fprintf(stderr, "Off-diagonal entries must be positive\n");
            fprintf(stderr, "diagraph[%d,%d] = %d\n", i, j, digraph[i*n+j])
            exit(-1);
         }
      }
}  /* Read_digraph */


/*------------------------------------------------------------------
 * Function:  Print_digraph
 * Purpose:   Print the number of cities and the digraphrix of costs
 * Globals in:
 *    n:        number of cities
 *    digraph:  digraph of costs
 */
void Print_digraph(void) {
   int i, j;

   printf("Order = %d\n", n);
   printf("Matrix = \n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)
         printf("%2d ", digraph[i*n+j]);
      printf("\n");
   }
   printf("\n");
}  /* Print_digraph */


/*------------------------------------------------------------------
 * Function:    Depth_first_search
 * Purpose:     Recursively search for a least-cost tour
 * In arg:
 *    tour:     partial tour of cities visited so far.
 * Globals in:
 *    n:        total number of cities in the problem
 * Note:
 *    The input tour is modified during execution of search,
 *    but returned to its original state before returning.
 */
void Depth_first_search(tour_t tour) {
   city_t nbr;

#  ifdef DEBUG
   Print_tour(tour, "Entering DFS");
   printf("City count = %d\n", City_count(tour));
   printf("Tour cost = %d\n", Tour_cost(tour));
   printf("Call count = %ld\n\n", ++call_count);
#  endif

   if (City_count(tour) == n) {
      if (Best_tour(tour)) {
         Update_best_tour(tour);
#        ifdef DEBUG
         Print_tour(best_tour, "After Update_best_tour");
         printf("City count = %d\n", City_count(best_tour));
         printf("Tour cost = %d\n\n", Tour_cost(best_tour));
#        endif
      }
   } else {
      for (nbr = 1; nbr < n; nbr++)
         if (Feasible(tour, nbr)) {
            Add_city(tour, nbr);
            Depth_first_search(tour);
            Remove_last_city(tour);
         }
   }

#  ifdef DEBUG
   Print_tour(tour, "Returning from DFS");
   printf("\n");
#  endif
}  /* Depth_first_search */
```

```c

/*------------------------------------------------------------------
 * Function:    Best_tour
 * Purpose:     Determine whether addition of the hometown to the
 *              n-city input tour will lead to a best tour.
 * In arg:
 *    tour:     tour visiting all n cities
 * Ret val:
 *    TRUE if best tour, FALSE otherwise
 */
int Best_tour(tour_t tour) {
   cost_t cost_so_far = Tour_cost(tour);
   city_t last_city = Last_city(tour);

   if (cost_so_far + Cost(last_city, home_town) < Tour_cost(best_tour))
      return TRUE;
   else
      return FALSE;
}  /* Best_tour */

/*------------------------------------------------------------------
 * Function:    Update_best_tour
 * Purpose:     Replace the existing best tour with the input tour +
 *              hometown
 * In arg:
 *    tour:     tour that's visited all n-cities
 * Global out:
 *    best_tour:  the current best tour
 * Note:
 *    The input tour hasn't had the home_town added as the last
 *    city before the call to Update_best_tour.  So we call
 *    Add_city(best_tour, hometown) before returning.
 */
void Update_best_tour(tour_t tour) {
   Copy_tour(tour, best_tour);
   Add_city(best_tour, home_town);
}  /* Update_best_tour */


/*------------------------------------------------------------------
 * Function:   Copy_tour
 * Purpose:    Copy tour1 into tour2
 * In arg:
 *    tour1
 * Out arg:
 *    tour2
 */
void Copy_tour(tour_t tour1, tour_t tour2) {
   int i;

   for (i = 0; i <= n; i++)
     tour2->cities[i] =  tour1->cities[i];
   tour2->count = tour1->count;
   tour2->cost = tour1->cost;
}  /* Copy_tour */

/*------------------------------------------------------------------
 * Function:  Add_city
 * Purpose:   Add city to the end of tour
 * In arg:
 *    city
 * In/out arg:
 *    tour
 */
void Add_city(tour_t tour, city_t new_city) {
   city_t old_last_city = Last_city(tour);
   tour->cities[tour->count] = new_city;
   (tour->count)++;
   tour->cost += Cost(old_last_city,new_city);
}  /* Add_city */

/*------------------------------------------------------------------
 * Function:  Remove_last_city
 * Purpose:   Remove last city from end of tour
 * In/out arg:
 *    tour
 * Note:
 *    Function assumes there are at least two cities on the tour --
 *    i.e., the hometown in tour->cities[0] won't be removed.
 */
void Remove_last_city(tour_t tour) {
   city_t old_last_city = Last_city(tour);
   city_t new_last_city;

   tour->cities[tour->count-1] = NO_CITY;
   (tour->count)--;
   new_last_city = Last_city(tour);
   tour->cost -= Cost(new_last_city,old_last_city);
}  /* Remove_last_city */

/*------------------------------------------------------------------
 * Function:  Feasible
 * Purpose:   Check whether nbr could possibly lead to a better
 *            solution if it is added to the current tour.  The
 *            function checks whether nbr has already been visited
 *            in the current tour, and, if not, whether adding the
 *            edge from the current city to nbr will result in
 *            a cost less than the current best cost.
 * In args:   All
 * Global in:
```

```c
 *    best_tour
 * Return:    TRUE if the nbr can be added to the current tour.
 *            FALSE otherwise
 */
int Feasible(tour_t tour, city_t city) {
   city_t last_city = Last_city(tour);

   if (!Visited(tour, city) &&
        Tour_cost(tour) + Cost(last_city,city) < Tour_cost(best_tour))
      return TRUE;
   else
      return FALSE;
}  /* Feasible */


/*------------------------------------------------------------------
 * Function:   Visited
 * Purpose:    Use linear search to determine whether city has already
 *             been visited on the current tour.
 * In args:    All
 * Return val: TRUE if city has already been visited.
 *             FALSE otherwise
 */
int Visited(tour_t tour, city_t city) {
   int i;

   for (i = 0; i < City_count(tour); i++)
      if ( Tour_city(tour,i) == city ) return TRUE;
   return FALSE;
}  /* Visited */


/*------------------------------------------------------------------
 * Function:  Print_tour
 * Purpose:   Print a tour
 * In args:   All
 */
void Print_tour(tour_t tour, char* title) {
   int i;

   printf("%s:\n", title);
   for (i = 0; i < City_count(tour); i++)
      printf("%d ", Tour_city(tour,i));
   printf("\n");
}  /* Print_tour */

```