```
==> ./graph.c <==
/*
graph.c

Set of vertices and edges implementation.

Implementations for helper functions for graph construction and manipulation.

Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
*/
#include <stdlib.h>
#include <assert.h>
#include <limits.h>
#include "graph.h"
#include "utils.h"
#include "pq.h"

#define INITIALEDGES 32
#define MAXNUMVERTICES 20
#define MAXNUMEDGES 50

struct edge;

/* Definition of a graph. */
struct graph {
  int numVertices;
  int numEdges;
  int allocedEdges;
  struct edge **edgeList;
};

/* Definition of an edge. */
struct edge {
  int start;
  int end;
  int cost;
};

struct graph *newGraph(int numVertices){
  struct graph *g = (struct graph *) malloc(sizeof(struct graph));
  assert(g);
  /* Initialise edges. */
  g->numVertices = numVertices;
  g->numEdges = 0;
  g->allocedEdges = 0;
  g->edgeList = NULL;
  return g;
}

/* Adds an edge to the given graph. */
void addEdge(struct graph *g, int start, int end, int cost){
  assert(g);
```

```
   53:   struct edge *newEdge = NULL;
   54:   /* Check we have enough space for the new edge. */
   55:   if((g->numEdges + 1) > g->allocedEdges){
   56:     if(g->allocedEdges == 0){
   57:       g->allocedEdges = INITIALEDGES;
   58:     } else {
   59:       (g->allocedEdges) *= 2;
   60:     }
   61:     g->edgeList = (struct edge **) realloc(g->edgeList,
   62:       sizeof(struct edge *) * g->allocedEdges);
   63:     assert(g->edgeList);
   64:   }
   65:
   66:   /* Create the edge */
   67:   newEdge = (struct edge *) malloc(sizeof(struct edge));
   68:   assert(newEdge);
   69:   newEdge->start = start;
   70:   newEdge->end = end;
   71:   newEdge->cost = cost;
   72:
   73:   /* Add the edge to the list of edges. */
   74:   g->edgeList[g->numEdges] = newEdge;
   75:   (g->numEdges)++;
   76: }
   77:
   78: /* Frees all memory used by graph. */
   79: void freeGraph(struct graph *g){
   80:   int i;
   81:   for(i = 0; i < g->numEdges; i++){
   82:     free((g->edgeList)[i]);
   83:   }
   84:   if(g->edgeList){
   85:     free(g->edgeList);
   86:   }
   87:   free(g);
   88: }
   89:
   90: struct solution *graphSolve(struct graph *g, enum problemPart part,
   91:   int antennaCost, int numHouses){
   92:   int cost[MAXNUMVERTICES];
   93:   int vertices[MAXNUMVERTICES];
   94:   int i;
   95:
   96:   struct solution *solution = (struct solution *)
   97:     malloc(sizeof(struct solution));
   98:   assert(solution);
   99:   if(part == PART_A){
  100:     /* IMPLEMENT 2A SOLUTION HERE */
  101:     solution->antennaTotal = antennaCost * numHouses;
  102:     solution->cableTotal = 0;
  103:
  104:     /* perform a form of Prim's Algorithm on the graph to find the mini
mum
```

```
  105:     spanning tree total cost and thus the solution to the cableTotal */
  106:
  107:     /* initialise costs to infinity (MAX_INT in C) */
  108:     for (i=0; i < numHouses + 1; i++){
  109:       cost[i] = INT_MAX;
  110:     }
  111:     /* initialise array containing vertex values */
  112:     for (i=0; i < numHouses + 1; i++){
  113:       vertices[i] = i;
  114:     }
  115:     cost[0] = 0;
  116:
  117:     /* initialise priority queue */
  118:     struct pq *pq = newPQ();
  119:     for (i=0; i < numHouses + 1; i++){
  120:       enqueue(pq, (void*)&(vertices[i]), cost[i]);
  121:     }
  122:
  123:     while (!empty(pq)){
  124:       int u = *(int*)deletemin(pq);
  125:       solution->cableTotal += cost[u];
  126:       /* for each (u,w) contained in edgeList, check if w is still in t
he
  127:       priority queue and if the weight(u, w) < cost[w] */
  128:       for (i=0; i < g->numEdges; i++){
  129:         if ((g->edgeList)[i]->start == u){
  130:           int w = (g->edgeList)[i]->end;
  131:           if (isinPQ(pq, &w) && (g->edgeList)[i]->cost < cost[w]){
  132:             cost[w] = (g->edgeList)[i]->cost;
  133:             updatePQ(pq, &w, cost[w]);
  134:           }
  135:         /* check the other direction as this is an undirected graph */
  136:         } else if ((g->edgeList)[i]->end == u){
  137:           int w = (g->edgeList)[i]->start;
  138:           if (isinPQ(pq, &w) && (g->edgeList)[i]->cost < cost[w]){
  139:             cost[w] = (g->edgeList)[i]->cost;
  140:             updatePQ(pq, &w, cost[w]);
  141:           }
  142:         }
  143:       }
  144:     }
  145:   } else {
  146:     /* IMPLEMENT 2C SOLUTION HERE */
  147:
  148:     /* can utilise the exact same algorithm as in part 2a with the only
  149:     difference being that we set the inital costs of all vertices to th
e
  150:     antenna cost as we can now choose to use an antenna when a cable co
nnection
  151:     would be more expensive */
  152:     solution->mixedTotal = 0;
  153:
  154:     /* initialise costs to antennaCost */
```

```
  155:     for (i=0; i < numHouses + 1; i++){
  156:       cost[i] = antennaCost;
  157:     }
  158:     /* initialise array containing vertex values */
  159:     for (i=0; i < numHouses + 1; i++){
  160:       vertices[i] = i;
  161:     }
  162:     cost[0] = 0;
  163:
  164:     /* initialise priority queue */
  165:     struct pq *pq = newPQ();
  166:     for (i=0; i < numHouses + 1; i++){
  167:       enqueue(pq, (void*)&(vertices[i]), cost[i]);
  168:     }
  169:
  170:     while (!empty(pq)){
  171:       int u = *(int*)deletemin(pq);
  172:       solution->mixedTotal += cost[u];
  173:       /* for each (u,w) contained in edgeList, check if w is still in t
he
  174:       priority queue and if the weight(u, w) < cost[w] */
  175:       for (i=0; i < g->numEdges; i++){
  176:         if ((g->edgeList)[i]->start == u){
  177:           int w = (g->edgeList)[i]->end;
  178:           if (isinPQ(pq, &w) && (g->edgeList)[i]->cost < cost[w]){
  179:             cost[w] = (g->edgeList)[i]->cost;
  180:             updatePQ(pq, &w, cost[w]);
  181:           }
  182:         /* check the other direction as this is an undirected graph */
  183:         } else if ((g->edgeList)[i]->end == u){
  184:           int w = (g->edgeList)[i]->start;
  185:           if (isinPQ(pq, &w) && (g->edgeList)[i]->cost < cost[w]){
  186:             cost[w] = (g->edgeList)[i]->cost;
  187:             updatePQ(pq, &w, cost[w]);
  188:           }
  189:         }
  190:       }
  191:     }
  192:   }
  193:   return solution;
  194: }
  195:
  196: ==> ./graph.h <==
  197: /*
  198: graph.h
  199:
  200: Visible structs and functions for graph construction and manipulation.
  201:
  202: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  203: */
  204:
  205: /* Definition of a graph. */
  206: struct graph;
```

```
  207:
  208: enum problemPart;
  209:
  210: struct solution;
  211:
  212: /* A particular solution to a graph problem. */
  213: #ifndef SOLUTION_STRUCT
  214: #define SOLUTION_STRUCT
  215: struct solution {
  216:   int antennaTotal;
  217:   int cableTotal;
  218:   int mixedTotal;
  219: };
  220: #endif
  221:
  222: /* Which part the program should find a solution for. */
  223: #ifndef PART_ENUM
  224: #define PART_ENUM
  225: enum problemPart {
  226:   PART_A=0,
  227:   PART_C=1
  228: };
  229: #endif
  230:
  231: /* Creates an undirected graph with the given numVertices and no edges
and
  232: returns a pointer to it. NumEdges is the number of expected edges. */
  233: struct graph *newGraph(int numVertices);
  234:
  235: /* Adds an edge to the given graph. */
  236: void addEdge(struct graph *g, int start, int end, int cost);
  237:
  238: /* Find the total radio-based cost, total cabled cost if the part is PA
RT_A, and
  239:   the mixed total cost if the part is PART_C. */
  240: struct solution *graphSolve(struct graph *g, enum problemPart part,
  241:   int antennaCost, int numHouses);
  242:
  243: /* Frees all memory used by graph. */
  244: void freeGraph(struct graph *g);
  245:
  246: /* Frees all data used by solution. */
  247: void freeSolution(struct solution *solution);
  248:
  249:
  250:
  251: ==> ./list.c <==
  252: /*
  253: list.c
  254:
  255: Implementations for helper functions for linked list construction and
  256: manipulation.
  257:
```

```
 258: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
 259: */
 260: #include "list.h"
 261: #include <stdlib.h>
 262: #include <assert.h>
 263:
 264: struct list {
 265:   void *item;
 266:   struct list *next;
 267: };
 268:
 269: struct list *newlist(void *item){
 270:   struct list *head = (struct list *) malloc(sizeof(struct list));
 271:   assert(head);
 272:   head->item = item;
 273:   head->next = NULL;
 274:   return head;
 275: }
 276:
 277: struct list *prependList(struct list *list, void *item){
 278:   struct list *head = (struct list *) malloc(sizeof(struct list));
 279:   assert(head);
 280:   head->item = item;
 281:   head->next = list;
 282:   return head;
 283: }
 284:
 285: void *peekHead(struct list *list){
 286:   if(! list){
 287:     return NULL;
 288:   }
 289:   return list->item;
 290: }
 291:
 292: void *deleteHead(struct list **list){
 293:   void *item;
 294:   struct list *next;
 295:   if(! list || ! *list){
 296:     return NULL;
 297:   }
 298:   /* Store values we're interested in before freeing list node. */
 299:   item = (*list)->item;
 300:   next = (*list)->next;
 301:   free(*list);
 302:   *list = next;
 303:   return item;
 304: }
 305:
 306: void freeList(struct list *list){
 307:   struct list *next;
 308:   /* Iterate through list until the end of the list (NULL) is reached.
*/
 309:   for(next = list; list != NULL; list = next){
```

```
  310:     /* Store next pointer before we free list's space. */
  311:     next = list->next;
  312:     free(list);
  313:   }
  314: }
  315: ==> ./list.h <==
  316: /*
  317: list.h
  318:
  319: Visible structs and functions for linked lists.
  320:
  321: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  322: */
  323: /* The linked list. */
  324: struct list;
  325:
  326: /* Get a new empty list. */
  327: struct list *newlist(void *item);
  328:
  329: /* Add an item to the head of the list. Returns the new list. */
  330: struct list *prependList(struct list *list, void *item);
  331:
  332: /* Gets the first item from the list. */
  333: void *peekHead(struct list *list);
  334:
  335: /* Takes the first item from the list, updating the list pointer and re
turns
  336:   the item stored. */
  337: void *deleteHead(struct list **list);
  338:
  339: /* Free all list items. */
  340: void freeList(struct list *list);
  341: ==> ./Makefile <==
  342: # Build targets
  343: # lm - link math library library. required if you use math.h functions
(commonly
  344: # linked by default on mac).
  345: problem2a: problem2a.o utils.o graph.o pq.o list.o
  346:     gcc -Wall -o problem2a -g -lm problem2a.o utils.o graph.o pq.o list
.o
  347:
  348: problem2c: problem2c.o utils.o graph.o pq.o list.o
  349:     gcc -Wall -o problem2c -g -lm problem2c.o utils.o graph.o pq.o list
.o
  350:
  351: problem3: problem3.o
  352:     gcc -Wall -o problem3 -g -lm problem3.o
  353:
  354:
  355: problem2a.o: problem2a.c graph.h utils.h
  356:     gcc -c problem2a.c -Wall -g
  357:
  358: problem2c.o: problem2c.c graph.h utils.h
```

```
  359:     gcc -c problem2c.c -Wall -g
  360:
  361: problem3.o: problem3.c
  362:     gcc -c problem3.c -Wall -g
  363:
  364: utils.o: utils.c utils.h graph.h
  365:     gcc -c utils.c -Wall -g
  366:
  367: graph.o: graph.c graph.h pq.h utils.h
  368:     gcc -c graph.c -Wall -g
  369:
  370: pq.o: pq.c pq.h
  371:     gcc -c pq.c -Wall -g
  372:
  373: list.o: list.c list.h
  374:     gcc -c list.c -Wall -g
  375:
  376: .PHONY: clean
  377:
  378: clean:
  379:     rm *.o *.exe==> ./pq.c <==
  380: /*
  381: pq.c
  382:
  383: Unsorted Array Implementation
  384:
  385: Implementations for helper functions for priority queue construction an
d
  386: manipulation.
  387:
  388: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  389: */
  390: #include <stdlib.h>
  391: #include <stdio.h>
  392: #include <assert.h>
  393: #include "pq.h"
  394:
  395: #define INITIALITEMS 32
  396: #define TRUE 1
  397: #define FALSE 0
  398:
  399: struct pq {
  400:   int count;
  401:   int allocated;
  402:   void **queue;
  403:   int *priorities;
  404: };
  405:
  406:
  407: struct pq *newPQ(){
  408:   struct pq *pq = (struct pq *) malloc(sizeof(struct pq));
  409:   assert(pq);
  410:   pq->count = 0;
```

```
  411:   pq->allocated = 0;
  412:   pq->queue = NULL;
  413:   pq->priorities = NULL;
  414:   return pq;
  415: }
  416:
  417: void enqueue(struct pq *pq, void *item, int priority){
  418:   assert(pq);
  419:   if((pq->count + 1) > pq->allocated){
  420:     if (pq->allocated == 0){
  421:       pq->allocated = INITIALITEMS;
  422:     } else {
  423:       pq->allocated *= 2;
  424:     }
  425:     pq->queue = (void **) realloc(pq->queue, pq->allocated * sizeof(voi
d *));
  426:     assert(pq->queue);
  427:     pq->priorities = (int *) realloc(pq->priorities, pq->allocated *
  428:       sizeof(int));
  429:     assert(pq->priorities);
  430:   }
  431:   (pq->queue)[pq->count] = item;
  432:   (pq->priorities)[pq->count] = priority;
  433:   (pq->count)++;
  434: }
  435:
  436: /* Scan through all the priorities linearly and find lowest. */
  437: void *deletemin(struct pq *pq){
  438:   int i;
  439:   int lowestElement = 0;
  440:   void *returnVal;
  441:   if (pq->count <= 0){
  442:     return NULL;
  443:   }
  444:   for(i = 0; i < pq->count; i++){
  445:       if((pq->priorities)[i] < (pq->priorities)[lowestElement]){
  446:       lowestElement = i;
  447:     }
  448:   }
  449:   returnVal = (pq->queue)[lowestElement];
  450:   /* Delete item from queue by swapping final item into place of delete
d
  451:     element. */
  452:   if(pq->count > 0){
  453:     (pq->priorities)[lowestElement] = (pq->priorities)[pq->count - 1];
  454:     (pq->queue)[lowestElement] = (pq->queue)[pq->count - 1];
  455:     (pq->count)--;
  456:   }
  457:   return returnVal;
  458: }
  459:
  460: int empty(struct pq *pq){
  461:   return pq->count == 0;
```

```
462: }
463:
464: /* find out whether an item is in the queue or not */
465: int isinPQ(struct pq *pq, void *item){
466:   int i;
467:
468:   for (i=0; i < pq->count; i++){
469:     if (*(int*)((pq->queue)[i]) == *(int*)item){
470:       return TRUE;
471:     }
472:    }
473:
474:    return FALSE;
475: }
476:
477: /* update the cost of a vertex in the priority queue */
478: void updatePQ(struct pq *pq, void *item, int newPriority) {
479:   int i;
480:
481:   /* find the vertex in the queue that we are changing the cost of */
482:   for (i=0; i < pq->count; i++){
483:     if (*(int*)((pq->queue)[i]) == *(int*)item){
484:       pq->priorities[i] = newPriority;
485:       break;
486:     }
487:   }
488: }
489:
490: void freePQ(struct pq *pq){
491:   if(! pq) {
492:     return;
493:   }
494:   if(pq->allocated > 0){
495:     free(pq->queue);
496:     free(pq->priorities);
497:   }
498:   free(pq);
499: }
500: ==> ./pq.h <==
501: /*
502: pq.h
503:
504: Visible structs and functions for priority queues.
505:
506: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
507: */
508: /* The priority queue. */
509: struct pq;
510:
511: /* Get a new empty priority queue. */
512: struct pq *newPQ();
513:
514: /* Add an item to the priority queue - cast pointer to (void *). */
```

```
  515: void enqueue(struct pq *pq, void *item, int priority);
  516:
  517: /* Take the smallest item from the priority queue - cast pointer back t
o
  518:   original type. */
  519: void *deletemin(struct pq *pq);
  520:
  521: /* Returns 1 if empty, 0 otherwise. */
  522: int empty(struct pq *pq);
  523:
  524: /* find out whether the item is in the queue or not */
  525: int isinPQ(struct pq *pq, void *item);
  526:
  527: /* update the cost of a vertex in the priority queue */
  528: void updatePQ(struct pq *pq, void *item, int newPriority);
  529:
  530: /* Remove all items from priority queue (doesn't free) and free the que
ue. */
  531: void freePQ(struct pq *pq);
  532: ==> ./problem2a.c <==
  533: /*
  534: problem2a.c
  535:
  536: Driver function for Problem 2 Part A.
  537:
  538: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  539: */
  540: #include <stdio.h>
  541: #include "utils.h"
  542: #include "graph.h"
  543:
  544: int main(int argc, char **argv){
  545:   /* Read the problem in from stdin. */
  546:   struct graphProblem *problem = readProblem(stdin);
  547:   /* Find the solution to the problem. */
  548:   struct solution *solution = findSolution(problem, PART_A);
  549:
  550:   /* Report solution */
  551:   // printf("Cost of installation using antennas %d\n", solution->anten
naTotal);
  552:   // printf("Cost of installation using cables %d\n", solution->cableTo
tal);
  553:
  554:   /* Print better choice. */
  555:   if(solution->cableTotal < solution->antennaTotal){
  556:     /* printf("Cheapest technology: Cabled installation cheapest\n"); *
/
  557:     printf("c\n");
  558:   } else if (solution->cableTotal == solution->antennaTotal){
  559:     /* printf("Cheapest technology: Both technologies equal cost\n"); *
/
  560:     printf("b\n");
  561:   } else {
```

```
  562:     /* printf("Cheapest technology: Radio-based installation cheapest\n
");  */
  563:     printf("r\n");
  564:   }
  565:
  566:   freeProblem(problem);
  567:   freeSolution(solution);
  568:
  569:   return 0;
  570: }
  571:
  572: ==> ./problem2c.c <==
  573: /*
  574: problem2c.c
  575:
  576: Driver function for Problem 2 Part C.
  577:
  578: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  579: */
  580: #include <stdio.h>
  581: #include "utils.h"
  582:
  583: int main(int argc, char **argv){
  584:   /* Read the problem in from stdin. */
  585:   struct graphProblem *problem = readProblem(stdin);
  586:   /* Find the solution to the problem. */
  587:   struct solution *solution = findSolution(problem, PART_C);
  588:
  589:   /* Report solution */
  590:   /* printf("Cost of installation using mixed technologies %d\n",
  591:     solution->mixedTotal); */
  592:   printf("%d\n", solution->mixedTotal);
  593:
  594:   freeProblem(problem);
  595:   freeSolution(solution);
  596:
  597:   return 0;
  598: }
  599:
  600: ==> ./problem3.c <==
  601: /*
  602: problem3.c
  603:
  604: Driver function for Problem 3.
  605:
  606: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  607: */
  608: #include <stdio.h>
  609: #include <stdlib.h>
  610: #include <assert.h>
  611: #include <math.h>
  612: #include <limits.h>
  613:
```

```
  614: /* Constants */
  615: #define OLDCHIP 0
  616: #define NEWCHIP 1
  617: #define MAXNUMERATOR 100
  618: #define MAXDENOMINATOR 100
  619:
  620: /* Used to store all the statistics for a single chip. */
  621: struct statistics;
  622:
  623: /* Used to store all the statistics for both chips for each problem. */
  624: struct chipStatistics;
  625:
  626: struct statistics {
  627:   int operations;
  628:   int instances;
  629:   int minOperations;
  630:   double avgOperations;
  631:   int maxOperations;
  632: };
  633:
  634: struct chipStatistics {
  635:   struct statistics oldChipEuclid;
  636:   struct statistics newChipEuclid;
  637:   struct statistics oldChipSieve;
  638:   struct statistics newChipSieve;
  639: };
  640:
  641: /* Set all statistics to 0s */
  642: void initialiseStatistics(struct statistics *stats);
  643:
  644: /* Collects the minimum, average and maximum operations from running al
l
  645: combinations of numerators from 1 to the given maxNumerator and 1 to th
e given
  646: maxDenominator. */
  647: void collectStatistics(struct chipStatistics *chipStats, int maxNumerat
or,
  648:   int maxDenominator);
  649:
  650: /* Divides the number of operations by the number of instances. */
  651: void calculateAverage(struct statistics *stats);
  652:
  653: /* Prints out the minimum, average and maximum operations from given
  654: statistics. */
  655: void printStatistics(struct statistics *stats);
  656:
  657: /* Calculates the number of operations required for Euclid's algorithm
given the
  658: numerator and denominator when running on the given chip type (one of O
LDCHIP
  659: and NEWCHIP) by moving through the steps of the algorithm and counting
each
  660: pseudocode operation. */
```

```
  661: void euclid(int numerator, int denominator, int chip, struct statistics
 *s);
  662:
  663: /* Calculates the number of operations required for the sieve of Eratos
thenes
  664: given the numerator and denominator when running on the given chip type
 (one of
  665: OLDCHIP and NEWCHIP) by moving through the steps of the algorithm and c
ounting
  666: each pseudocode operation. */
  667: void eratosthenes(int numerator, int denominator, int chip,
  668:   struct statistics *s);
  669:
  670: int main(int argc, char **argv){
  671:   struct chipStatistics summaryStatistics;
  672:
  673:   collectStatistics(&summaryStatistics, MAXNUMERATOR, MAXDENOMINATOR);
  674:
  675:   printf("Old chip (Euclid):\n");
  676:   printStatistics(&(summaryStatistics.oldChipEuclid));
  677:   printf("\n");
  678:   printf("New chip (Euclid)\n");
  679:   printStatistics(&(summaryStatistics.newChipEuclid));
  680:   printf("\n");
  681:   printf("Old chip (Sieve)\n");
  682:   printStatistics(&(summaryStatistics.oldChipSieve));
  683:   printf("\n");
  684:   printf("New chip (Sieve)\n");
  685:   printStatistics(&(summaryStatistics.newChipSieve));
  686:   printf("\n");
  687:
  688:   return 0;
  689: }
  690:
  691: void collectStatistics(struct chipStatistics *chipStats, int maxNumerat
or,
  692:   int maxDenominator){
  693:   int numerator, denominator;
  694:   /* Initialise all statistics */
  695:   initialiseStatistics(&(chipStats->oldChipEuclid));
  696:   initialiseStatistics(&(chipStats->newChipEuclid));
  697:   initialiseStatistics(&(chipStats->oldChipSieve));
  698:   initialiseStatistics(&(chipStats->newChipSieve));
  699:
  700:   for(numerator = 1; numerator <= maxNumerator; numerator++){
  701:     for(denominator = 1; denominator <= maxDenominator; denominator++){
  702:       /* Run algorithms for all combinations of numerator and denominat
or. */
  703:       euclid(numerator, denominator, OLDCHIP,
  704:         &(chipStats->oldChipEuclid));
  705:       euclid(numerator, denominator, NEWCHIP,
  706:         &(chipStats->newChipEuclid));
  707:       eratosthenes(numerator, denominator, OLDCHIP,
```

```
  708:         &(chipStats->oldChipSieve));
  709:       eratosthenes(numerator, denominator, NEWCHIP,
  710:         &(chipStats->newChipSieve));
  711:     }
  712:   }
  713:   calculateAverage(&(chipStats->oldChipEuclid));
  714:   calculateAverage(&(chipStats->newChipEuclid));
  715:   calculateAverage(&(chipStats->oldChipSieve));
  716:   calculateAverage(&(chipStats->newChipSieve));
  717: }
  718:
  719: void calculateAverage(struct statistics *stats){
  720:   stats->avgOperations = (double) stats->operations / stats->instances;
  721: }
  722:
  723: void initialiseStatistics(struct statistics *stats){
  724:   stats->operations = 0;
  725:   stats->instances = 0;
  726:   stats->minOperations = INT_MAX;
  727:   stats->avgOperations = 0;
  728:   stats->maxOperations = 0;
  729: }
  730:
  731: void euclid(int numerator, int denominator, int chip, struct statistics
*s){
  732:   /* IMPLEMENT THIS */
  733:   int currOperations = 0;
  734:   int temp;
  735:
  736:   s->instances += 1;
  737:   int b = numerator;
  738:   int a = denominator;
  739:   currOperations += 2; // 2 assignments
  740:   while (b != 0){
  741:     temp = b;
  742:     b = a%b;
  743:     a = temp;
  744:     currOperations += 9; // 1 mod, 3 assignment, 1 while check
  745:   }
  746:   currOperations += 1; // include the last check of the while loop
  747:   // printf("%d / %d\n", (numerator/a), (denominator/a));
  748:   currOperations += 10; // 2 division, print is free
  749:
  750:   s->operations += currOperations;
  751:   /* check if this instance required the least amount or largest amount
of
  752:   operations so far */
  753:   if (currOperations < s->minOperations){
  754:       s->minOperations = currOperations;
  755:   } else if (currOperations > s->maxOperations){
  756:       s->maxOperations = currOperations;
  757:   }
  758: }
```

```
  759:
  760: void eratosthenes(int numerator, int denominator, int chip,
  761:   struct statistics *s){
  762:   /* IMPLEMENT THIS */
  763:   int i, j;
  764:   int currOperations = 0;
  765:
  766:   s->instances += 1;
  767:   int num = numerator;
  768:   int den = denominator;
  769:   currOperations += 2; // 2 assignments
  770:
  771:   /* numCandidates = min(num, den) */
  772:   int numCandidates;
  773:   if (num <= den){
  774:     numCandidates = num;
  775:     currOperations += 1; // 1 assignment
  776:   } else{
  777:       numCandidates = den;
  778:       currOperations += 1; // 1 assignment
  779:   }
  780:
  781:    /* initialise the primes array */
  782:   int primes[MAXNUMERATOR];
  783:   i = 1;
  784:   while (i < numCandidates + 1){
  785:     primes[i] = 1;
  786:     i += 1;
  787:   }
  788:   currOperations += 1; // primes assignment
  789:
  790:   i = 1;
  791:   currOperations += 1;
  792:   while (i < numCandidates){
  793:     i += 1;
  794:     currOperations += 3; // 1 assignment, 1 while check, 1 if check bel
ow
  795:     if (primes[i]){
  796:
  797:       j = i + i;  // start of line 13
  798:       currOperations += 1 * !chip; // 1 assignment on old chip
  799:       while (j <= numCandidates){
  800:           currOperations += 1 * !chip; // 1 while check on old chip
  801:         if ((j / i) > 1 && j%i == 0){
  802:           primes[j] = 0;
  803:           currOperations += 1 * !chip; // 1 assignment on old chip
  804:         }
  805:         j += i;
  806:         currOperations += 12 * !chip; // 1 division, 1 mod, 1 if, 1 ass
ignment
  807:                                       // on old chip
  808:       }                       // end of line 13
  809:       currOperations += 1; // include the last check of the while loop
```

```
  810:                            // or for the new chip this is the cost of l
ine 13
  811:
  812:         while (num%i == 0 && den%i == 0){
  813:           num = num / i;
  814:           den = den / i;
  815:           currOperations += 23;  // 2 mod, 2 division, 2 assignment, 1 wh
ile
  816:         }
  817:         currOperations += 11; // include the last check of the while loop
  818:       }
  819:     }
  820:     currOperations += 1; // include the last check of the while loop
  821:     // printf("%d / %d\n", num, den);
  822:
  823:     s->operations += currOperations;
  824:     /* check if this instance required the least amount or largest amount
 of
  825:     operations so far */
  826:     if (currOperations < s->minOperations){
  827:         s->minOperations = currOperations;
  828:     } else if (currOperations > s->maxOperations){
  829:         s->maxOperations = currOperations;
  830:     }
  831:
  832: }
  833:
  834: void printStatistics(struct statistics *stats){
  835:   printf("Minimum operations: %d\n", stats->minOperations);
  836:   printf("Average operations: %f\n", stats->avgOperations);
  837:   printf("Maximum operations: %d\n", stats->maxOperations);
  838: }
  839:
  840: ==> ./utils.c <==
  841: /*
  842: utils.c
  843:
  844: Implementations for helper functions to do with reading and writing.
  845:
  846: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  847: */
  848: #include <stdio.h>
  849: #include <stdlib.h>
  850: #include <assert.h>
  851: #include "graph.h"
  852: #include "utils.h"
  853:
  854: struct graphProblem {
  855:   int antennaCost;
  856:   int numHouses;
  857:   int numConnections;
  858:   struct graph *graph;
  859: };
```

```
  860:
  861: struct graphProblem *readProblem(FILE *file){
  862:   int i;
  863:   int startHouse;
  864:   int endHouse;
  865:   int cost;
  866:   /* Allocate space for problem specification */
  867:   struct graphProblem *problem = (struct graphProblem *)
  868:     malloc(sizeof(struct graphProblem));
  869:   assert(problem);
  870:
  871:   /* First line of input is antenna cost. */
  872:   assert(scanf("%d", &(problem->antennaCost)) == 1);
  873:   /* Next line comprises number of houses and number of connections. */
  874:   assert(scanf("%d %d", &(problem->numHouses), &(problem->numConnection
s))
  875:     == 2);
  876:
  877:   /* Build graph number of houses + 1 because of datacentre. */
  878:   problem->graph = newGraph(problem->numHouses + 1);
  879:   /* Add all edges to graph. */
  880:   for(i = 0; i < problem->numConnections; i++){
  881:     assert(scanf("%d %d %d", &startHouse, &endHouse, &cost) == 3);
  882:     addEdge(problem->graph, startHouse, endHouse, cost);
  883:   }
  884:
  885:   return problem;
  886: }
  887:
  888: struct solution *findSolution(struct graphProblem *problem,
  889:   enum problemPart part){
  890:   return graphSolve(problem->graph, part, problem->antennaCost,
  891:     problem->numHouses);
  892: }
  893:
  894: void freeProblem(struct graphProblem *problem){
  895:   /* No need to free if no data allocated. */
  896:   if(! problem){
  897:     return;
  898:   }
  899:   freeGraph(problem->graph);
  900:   free(problem);
  901: }
  902:
  903: void freeSolution(struct solution *solution){
  904:   /* No need to free if no data allocated. */
  905:   if(! solution){
  906:     return;
  907:   }
  908:   free(solution);
  909: }
  910: ==> ./utils.h <==
  911: /*
```

```
  912: utils.h
  913:
  914: Visible structs and functions for helper functions to do with reading a
nd
  915: writing.
  916:
  917: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  918: */
  919: /* Because we use FILE in this file, we should include stdio.h here. */
  920: #include <stdio.h>
  921: /* Because we use struct graph in this file, we should include graph.h
here. */
  922: #include "graph.h"
  923: /* The problem specified. */
  924: struct graphProblem;
  925:
  926: /* Reads the data from the given file pointer and returns a pointer to
this
  927: information. */
  928: struct graphProblem *readProblem(FILE *file);
  929:
  930: /* Finds a solution for a given problem. */
  931: struct solution *findSolution(struct graphProblem *problem,
  932:   enum problemPart part);
  933:
  934: /* Frees all data used by problem. */
  935: void freeProblem(struct graphProblem *problem);
  936:
```