```
==> ./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 <math.h>
#include "graph.h"
#include "utils.h"
#include "pq.h"

#define INITIALEDGES 32

struct edge;

/* Helper function to update edges if a new lower cost edge for that vertice
exist */
void updateLYNKED(struct graph *g, int curr, struct edge *LYNKED[]);

/* Helper function to initialise an array of "struct edge pointers" */
void initLYNKEDLINKED(struct graph *g, int LINKED[],
        struct edge *LYNKED[]);

/* Helper function to feed relevent edges into the priority queue */
void enqueupdate(struct graph *g, struct pq *prique,
        int LINKED[], struct edge *LYNKED[]);

/* 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));
```

```
  52:   assert(g);
  53:   /* Initialise edges. */
  54:   g->numVertices = numVertices;
  55:   g->numEdges = 0;
  56:   g->allocedEdges = 0;
  57:   g->edgeList = NULL;
  58:   return g;
  59: }
  60:
  61: /* Adds an edge to the given graph. */
  62: void addEdge(struct graph *g, int start, int end, int cost){
  63:   assert(g);
  64:   struct edge *newEdge = NULL;
  65:   /* Check we have enough space for the new edge. */
  66:   if((g->numEdges + 1) > g->allocedEdges){
  67:     if(g->allocedEdges == 0){
  68:       g->allocedEdges = INITIALEDGES;
  69:     } else {
  70:       (g->allocedEdges) *= 2;
  71:     }
  72:     g->edgeList = (struct edge **) realloc(g->edgeList,
  73:       sizeof(struct edge *) * g->allocedEdges);
  74:     assert(g->edgeList);
  75:   }
  76:
  77:   /* Create the edge */
  78:   newEdge = (struct edge *) malloc(sizeof(struct edge));
  79:   assert(newEdge);
  80:   newEdge->start = start;
  81:   newEdge->end = end;
  82:   newEdge->cost = cost;
  83:
  84:   /* Add the edge to the list of edges. */
  85:   g->edgeList[g->numEdges] = newEdge;
  86:   (g->numEdges)++;
  87: }
  88:
  89: /* Frees all memory used by graph. */
  90: void freeGraph(struct graph *g){
  91:   int i;
  92:   for(i = 0; i < g->numEdges; i++){
  93:     free((g->edgeList)[i]);
  94:   }
  95:   if(g->edgeList){
  96:     free(g->edgeList);
  97:   }
  98:   free(g);
  99: }
 100:
 101: /* Helper function which finds the solution to the problems */
 102: struct solution *graphSolve(struct graph *g, enum problemPart part,
 103:   int antennaCost, int numHouses){
 104:
```

```
  105:     /* shared variables between two problems */
  106:     int LINKED[g->numVertices];
  107:     int counter = 0, tally = 0, curr=0, i;
  108:     struct pq *prique;
  109:     struct edge *LYNKED[g->numVertices];
  110:     struct edge *temp;
  111:
  112:     /* variabels unique to part c */
  113:     int addantenna = 0;
  114:
  115:     struct solution *solution = (struct solution *)
  116:         malloc(sizeof(struct solution));
  117:     assert(solution);
  118:
  119:     if(part == PART_A){
  120:         /* IMPLEMENT 2A SOLUTION HERE */
  121:
  122:         /* initialise both arrays which help keep track */
  123:         initLYNKEDLINKED(g, LINKED, LYNKED);
  124:
  125:         /* use mechanism to find shortest path to each edge, and then s
elect
  126:         shortest path overall (prims' algorithm using given data struct
ures) */
  127:         while (counter < g->numVertices-1){
  128:             updateLYNKED(g, curr, LYNKED);
  129:
  130:             prique = newPQ();
  131:             enqueupdate(g, prique, LINKED, LYNKED);
  132:
  133:             temp = (struct edge*)deletemin(prique);
  134:             LINKED[temp->end] = 0;
  135:             tally += temp->cost;
  136:             curr = temp->end;
  137:             counter++;
  138:
  139:             free(prique);
  140:         }
  141:
  142:         /* free all the things that were created inside this function *
/
  143:         for (i=0; i<g->numVertices; i++){
  144:             free(LYNKED[i]);
  145:         }
  146:
  147:         /* realy the final results back to the calling function */
  148:         solution->antennaTotal = numHouses * antennaCost;
  149:         solution->cableTotal = tally;
  150:
  151:     } else {
  152:         /* IMPLEMENT 2C SOLUTION HERE */
  153:
  154:         /* initialise both arrays which help keep track */
```

```
  155:          initLYNKEDLINKED(g, LINKED, LYNKED);
  156:
  157:          /* use mechanism to find shortest path to each edge, and then s
elect
  158:          shortest path overall (prims' algorithm using given data struct
ures) */
  159:          while (counter < g->numVertices-1){
  160:              updateLYNKED(g, curr, LYNKED);
  161:
  162:              prique = newPQ();
  163:              enqueupdate(g, prique, LINKED, LYNKED);
  164:
  165:              temp = (struct edge*)deletemin(prique);
  166:              LINKED[temp->end] = 0;
  167:              curr = temp->end;
  168:              counter++;
  169:
  170:              /* add to cost only if the path cost less than the antenna,

  171:              otherwise more cheap to build an antenna at that house and
don't
  172:              build the path (but still keep traverse it as if we are doi
ng
  173:              prim's) */
  174:              if (temp->cost < antennaCost) {
  175:                  tally += temp->cost;
  176:              } else {
  177:                  addantenna++;
  178:              }
  179:
  180:              free(prique);
  181:
  182:          }
  183:
  184:          /* free all the things that were created inside this function *
/
  185:          for (i=0; i<g->numVertices; i++){
  186:              free(LYNKED[i]);
  187:          }
  188:
  189:          /* realy the final results back to the calling function */
  190:          solution->mixedTotal = tally + addantenna * antennaCost;
  191:      }
  192:      return solution;
  193: }
  194:
  195: /* Helper function to update edges if a new lower cost edge for that ve
rtice
  196: exist */
  197: void
  198: updateLYNKED(struct graph *g, int curr, struct edge *LYNKED[]){
  199:      int i;
  200:
```

```
  201:     /* walk through all edges, inefficient but does the job */
  202:     for (i=0; i<(g->numEdges); i++){
  203:         /* if either the start or end matches the curr vertex (as if do
ing
  204:         prim's algo manually) */
  205:         if (g->edgeList[i]->start == curr || g->edgeList[i]->end == cur
r){
  206:             /* two diff cases - with 'opposite' actions but ultimately
doing
  207:             the same job */
  208:             if (g->edgeList[i]->start == curr){
  209:                 /* if current cost is higher than this edge's cost */
  210:                 if (g->edgeList[i]->cost <
  211:                         LYNKED[g->edgeList[i]->end]->cost){
  212:
  213:                     LYNKED[g->edgeList[i]->end]->start = curr;
  214:                     LYNKED[g->edgeList[i]->end]->end = g->edgeList[i]->
end;
  215:                     LYNKED[g->edgeList[i]->end]->cost =
  216:                         g->edgeList[i]->cost;
  217:                 }
  218:             } else {
  219:                 /* if current cost is higher than this edge's cost */
  220:                 if (g->edgeList[i]->cost <
  221:                         LYNKED[g->edgeList[i]->start]->cost){
  222:
  223:                     LYNKED[g->edgeList[i]->start]->start = curr;
  224:                     LYNKED[g->edgeList[i]->start]->end =
  225:                         g->edgeList[i]->start;
  226:                     LYNKED[g->edgeList[i]->start]->cost =
  227:                         g->edgeList[i]->cost;
  228:                 }
  229:             }
  230:         }
  231:     }
  232: }
  233:
  234: /* Helper function to initialise an array of "struct edge pointers" */
  235: void
  236: initLYNKEDLINKED(struct graph *g, int LINKED[], struct edge *LYNKED[]){

  237:     int i;
  238:     struct edge *temp;
  239:
  240:     for (i=0; i<g->numVertices; i++){
  241:         temp = (struct edge *) malloc(sizeof(struct edge));
  242:         assert(temp);
  243:         temp->start = 0;
  244:         temp->end = 0;
  245:         temp->cost = INT_MAX; /* so that the first cost is always re
corded */
  246:         LYNKED[i] = temp;
  247:         LINKED[i] = 1; /* all verticies are yet to be connected */
```

```
  248:     }
  249:     LINKED[0] = 0;/*so that the data centre is not considered for conne
ction*/
  250: }
  251:
  252: /* Helper function to feed relevent edges into the priority queue */
  253: void
  254: enqueupdate(struct graph *g, struct pq *prique,
  255:         int LINKED[], struct edge *LYNKED[]){
  256:
  257:     int i;
  258:
  259:     /* add all verticies after the current round of prim's into the pri
ority
  260:     queue so the minimum of them can be outputted later */
  261:     for (i=0; i<g->numVertices; i++){
  262:         if (LINKED[i] == 1){
  263:             enqueue(prique, LYNKED[i], LYNKED[i]->cost);
  264:         }
  265:     }
  266: }==> ./graph.h <==
  267: /*
  268: graph.h
  269:
  270: Visible structs and functions for graph construction and manipulation.
  271:
  272: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  273: */
  274:
  275: /* Definition of a graph. */
  276: struct graph;
  277:
  278: enum problemPart;
  279:
  280: struct solution;
  281:
  282: /* A particular solution to a graph problem. */
  283: #ifndef SOLUTION_STRUCT
  284: #define SOLUTION_STRUCT
  285: struct solution {
  286:   int antennaTotal;
  287:   int cableTotal;
  288:   int mixedTotal;
  289: };
  290: #endif
  291:
  292: /* Which part the program should find a solution for. */
  293: #ifndef PART_ENUM
  294: #define PART_ENUM
  295: enum problemPart {
  296:   PART_A=0,
  297:   PART_C=1
  298: };
```

```
  299: #endif
  300:
  301: /* Creates an undirected graph with the given numVertices and no edges
and
  302: returns a pointer to it. NumEdges is the number of expected edges. */
  303: struct graph *newGraph(int numVertices);
  304:
  305: /* Adds an edge to the given graph. */
  306: void addEdge(struct graph *g, int start, int end, int cost);
  307:
  308: /* Find the total radio-based cost, total cabled cost if the part is PA
RT_A, and
  309:    the mixed total cost if the part is PART_C. */
  310: struct solution *graphSolve(struct graph *g, enum problemPart part,
  311:   int antennaCost, int numHouses);
  312:
  313: /* Frees all memory used by graph. */
  314: void freeGraph(struct graph *g);
  315:
  316: /* Frees all data used by solution. */
  317: void freeSolution(struct solution *solution);
  318:
  319:
  320:
  321:
  322:
  323: ==> ./list.c <==
  324: /*
  325: list.c
  326:
  327: Implementations for helper functions for linked list construction and
  328: manipulation.
  329:
  330: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  331: */
  332: #include "list.h"
  333: #include <stdlib.h>
  334: #include <assert.h>
  335:
  336: struct list {
  337:   void *item;
  338:   struct list *next;
  339: };
  340:
  341: struct list *newlist(void *item){
  342:   struct list *head = (struct list *) malloc(sizeof(struct list));
  343:   assert(head);
  344:   head->item = item;
  345:   head->next = NULL;
  346:   return head;
  347: }
  348:
  349: struct list *prependList(struct list *list, void *item){
```

```
  350:   struct list *head = (struct list *) malloc(sizeof(struct list));
  351:   assert(head);
  352:   head->item = item;
  353:   head->next = list;
  354:   return head;
  355: }
  356:
  357: void *peekHead(struct list *list){
  358:   if(! list){
  359:     return NULL;
  360:   }
  361:   return list->item;
  362: }
  363:
  364: void *deleteHead(struct list **list){
  365:   void *item;
  366:   struct list *next;
  367:   if(! list || ! *list){
  368:     return NULL;
  369:   }
  370:   /* Store values we're interested in before freeing list node. */
  371:   item = (*list)->item;
  372:   next = (*list)->next;
  373:   free(*list);
  374:   *list = next;
  375:   return item;
  376: }
  377:
  378: void freeList(struct list *list){
  379:   struct list *next;
  380:   /* Iterate through list until the end of the list (NULL) is reached.
*/
  381:   for(next = list; list != NULL; list = next){
  382:     /* Store next pointer before we free list's space. */
  383:     next = list->next;
  384:     free(list);
  385:   }
  386: }
  387: ==> ./list.h <==
  388: /*
  389: list.h
  390:
  391: Visible structs and functions for linked lists.
  392:
  393: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  394: */
  395: /* The linked list. */
  396: struct list;
  397:
  398: /* Get a new empty list. */
  399: struct list *newlist(void *item);
  400:
  401: /* Add an item to the head of the list. Returns the new list. */
```

```
  402: struct list *prependList(struct list *list, void *item);
  403:
  404: /* Gets the first item from the list. */
  405: void *peekHead(struct list *list);
  406:
  407: /* Takes the first item from the list, updating the list pointer and re
turns
  408:   the item stored. */
  409: void *deleteHead(struct list **list);
  410:
  411: /* Free all list items. */
  412: void freeList(struct list *list);
  413: ==> ./makefile <==
  414: # Build targets
  415: # lm - link math library library. required if you use math.h functions
(commonly
  416: # linked by default on mac).
  417: problem2a: problem2a.o utils.o graph.o pq.o list.o
  418:     gcc -Wall -o problem2a -g -lm problem2a.o utils.o graph.o pq.o list
.o
  419:
  420: problem2c: problem2c.o utils.o graph.o pq.o list.o
  421:     gcc -Wall -o problem2c -g -lm problem2c.o utils.o graph.o pq.o list
.o
  422:
  423: problem3: problem3.o
  424:     gcc -Wall -o problem3 -g -lm problem3.o
  425:
  426:
  427: problem2a.o: problem2a.c graph.h utils.h
  428:     gcc -c problem2a.c -Wall -g
  429:
  430: problem2c.o: problem2c.c graph.h utils.h
  431:     gcc -c problem2c.c -Wall -g
  432:
  433: problem3.o: problem3.c
  434:     gcc -c problem3.c -Wall -g
  435:
  436: utils.o: utils.c utils.h graph.h
  437:     gcc -c utils.c -Wall -g
  438:
  439: graph.o: graph.c graph.h pq.h utils.h
  440:     gcc -c graph.c -Wall -g
  441:
  442: pq.o: pq.c pq.h
  443:     gcc -c pq.c -Wall -g
  444:
  445: list.o: list.c list.h
  446:     gcc -c list.c -Wall -g
  447: ==> ./pq.c <==
  448: /*
  449: pq.c
  450:
```

```
  451: Unsorted Array Implementation
  452:
  453: Implementations for helper functions for priority queue construction an
d
  454: manipulation.
  455:
  456: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  457: */
  458: #include <stdlib.h>
  459: #include <assert.h>
  460:
  461: #define INITIALITEMS 32
  462:
  463: struct pq {
  464:   int count;
  465:   int allocated;
  466:   void **queue;
  467:   int *priorities;
  468: };
  469:
  470:
  471: struct pq *newPQ(){
  472:   struct pq *pq = (struct pq *) malloc(sizeof(struct pq));
  473:   assert(pq);
  474:   pq->count = 0;
  475:   pq->allocated = 0;
  476:   pq->queue = NULL;
  477:   pq->priorities = NULL;
  478:   return pq;
  479: }
  480:
  481: void enqueue(struct pq *pq, void *item, int priority){
  482:   assert(pq);
  483:   if((pq->count + 1) > pq->allocated){
  484:     if (pq->allocated == 0){
  485:       pq->allocated = INITIALITEMS;
  486:     } else {
  487:       pq->allocated *= 2;
  488:     }
  489:     pq->queue = (void **) realloc(pq->queue, pq->allocated * sizeof(voi
d *));
  490:     assert(pq->queue);
  491:     pq->priorities = (int *) realloc(pq->priorities, pq->allocated *
  492:       sizeof(int));
  493:     assert(pq->priorities);
  494:   }
  495:   (pq->queue)[pq->count] = item;
  496:   (pq->priorities)[pq->count] = priority;
  497:   (pq->count)++;
  498: }
  499:
  500: /* Scan through all the priorities linearly and find lowest. */
  501: void *deletemin(struct pq *pq){
```

```
  502:   int i;
  503:   int lowestElement = 0;
  504:   void *returnVal;
  505:   if (pq->count <= 0){
  506:     return NULL;
  507:   }
  508:   for(i = 0; i < pq->count; i++){
  509:     if((pq->priorities)[i] < (pq->priorities)[lowestElement]){
  510:       lowestElement = i;
  511:     }
  512:   }
  513:   returnVal = (pq->queue)[lowestElement];
  514:   /* Delete item from queue by swapping final item into place of delete
d
  515:     element. */
  516:   if(pq->count > 0){
  517:     (pq->priorities)[lowestElement] = (pq->priorities)[pq->count - 1];
  518:     (pq->queue)[lowestElement] = (pq->queue)[pq->count - 1];
  519:     (pq->count)--;
  520:   }
  521:   return returnVal;
  522: }
  523:
  524: int empty(struct pq *pq){
  525:   return pq->count == 0;
  526: }
  527:
  528: void freePQ(struct pq *pq){
  529:   if(! pq) {
  530:     return;
  531:   }
  532:   if(pq->allocated > 0){
  533:     free(pq->queue);
  534:     free(pq->priorities);
  535:   }
  536:   free(pq);
  537: }
  538: ==> ./pq.h <==
  539: /*
  540: pq.h
  541:
  542: Visible structs and functions for priority queues.
  543:
  544: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  545: */
  546: /* The priority queue. */
  547: struct pq;
  548:
  549: /* Get a new empty priority queue. */
  550: struct pq *newPQ();
  551:
  552: /* Add an item to the priority queue - cast pointer to (void *). */
  553: void enqueue(struct pq *pq, void *item, int priority);
```

```
  554:
  555: /* Take the smallest item from the priority queue - cast pointer back t
o
  556:   original type. */
  557: void *deletemin(struct pq *pq);
  558:
  559: /* Returns 1 if empty, 0 otherwise. */
  560: int empty(struct pq *pq);
  561:
  562: /* Remove all items from priority queue (doesn't free) and free the que
ue. */
  563: void freePQ(struct pq *pq);
  564: ==> ./problem2a.c <==
  565: /*
  566: problem2a.c
  567:
  568: Driver function for Problem 2 Part A.
  569:
  570: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  571: */
  572: #include <stdio.h>
  573: #include "utils.h"
  574: #include "graph.h"
  575:
  576: int main(int argc, char **argv){
  577:   /* Read the problem in from stdin. */
  578:   struct graphProblem *problem = readProblem(stdin);
  579:   /* Find the solution to the problem. */
  580:   struct solution *solution = findSolution(problem, PART_A);
  581:
  582:   /* Report solution */
  583:   /* printf("Cost of installation using antennas %d\n", solution->anten
naTotal); */
  584:   /* printf("Cost of installation using cables %d\n", solution->cableTo
tal); */
  585:
  586:   /* Print better choice. */
  587:   if(solution->cableTotal < solution->antennaTotal){
  588:     /* printf("Cheapest technology: Cabled installation cheapest\n"); *
/
  589:     printf("c\n");
  590:   } else if (solution->cableTotal == solution->antennaTotal){
  591:     /* printf("Cheapest technology: Both technologies equal cost\n"); *
/
  592:     printf("b\n");
  593:   } else {
  594:     /* printf("Cheapest technology: Radio-based installation cheapest\n
"); */
  595:     printf("r\n");
  596:   }
  597:
  598:   freeProblem(problem);
  599:   freeSolution(solution);
```

```
600:
601:   return 0;
602: }
603:
604: ==> ./problem2c.c <==
605: /*
606: problem2c.c
607:
608: Driver function for Problem 2 Part C.
609:
610: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
611: */
612: #include <stdio.h>
613: #include "utils.h"
614:
615: int main(int argc, char **argv){
616:   /* Read the problem in from stdin. */
617:   struct graphProblem *problem = readProblem(stdin);
618:   /* Find the solution to the problem. */
619:   struct solution *solution = findSolution(problem, PART_C);
620:
621:   /* Report solution */
622:   /* printf("Cost of installation using mixed technologies %d\n",
623:     solution->mixedTotal); */
624:   printf("%d\n", solution->mixedTotal);
625:
626:   freeProblem(problem);
627:   freeSolution(solution);
628:
629:   return 0;
630: }
631:
632: ==> ./problem3.c <==
633: /*
634: problem3.c
635:
636: Driver function for Problem 3.
637:
638: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
639: */
640: #include <stdio.h>
641: #include <stdlib.h>
642: #include <assert.h>
643: #include <math.h>
644: #include <limits.h>
645:
646: /* Constants */
647: #define OLDCHIP 0
648: #define NEWCHIP 1
649: #define MAXNUMERATOR 100
650: #define MAXDENOMINATOR 100
651:
652: /* Used to store all the statistics for a single chip. */
```

```
  653: struct statistics;
  654:
  655: /* Used to store all the statistics for both chips for each problem. */
  656: struct chipStatistics;
  657:
  658: struct statistics {
  659:   int operations;
  660:   int instances;
  661:   int minOperations;
  662:   double avgOperations;
  663:   int maxOperations;
  664: };
  665:
  666: struct chipStatistics {
  667:   struct statistics oldChipEuclid;
  668:   struct statistics newChipEuclid;
  669:   struct statistics oldChipSieve;
  670:   struct statistics newChipSieve;
  671: };
  672:
  673: /* Set all statistics to 0s */
  674: void initialiseStatistics(struct statistics *stats);
  675:
  676: /* Collects the minimum, average and maximum operations from running al
l
  677: combinations of numerators from 1 to the given maxNumerator and 1 to th
e given
  678: maxDenominator. */
  679: void collectStatistics(struct chipStatistics *chipStats, int maxNumerat
or,
  680:   int maxDenominator);
  681:
  682: /* Divides the number of operations by the number of instances. */
  683: void calculateAverage(struct statistics *stats);
  684:
  685: /* Prints out the minimum, average and maximum operations from given
  686: statistics. */
  687: void printStatistics(struct statistics *stats);
  688:
  689: /* Calculates the number of operations required for Euclid's algorithm
given the
  690: numerator and denominator when running on the given chip type (one of O
LDCHIP
  691: and NEWCHIP) by moving through the steps of the algorithm and counting
each
  692: pseudocode operation. */
  693: void euclid(int numerator, int denominator, int chip, struct statistics
 *s);
  694:
  695: /* Calculates the number of operations required for the sieve of Eratos
thenes
  696: given the numerator and denominator when running on the given chip type
 (one of
```

```
  697: OLDCHIP and NEWCHIP) by moving through the steps of the algorithm and c
ounting
  698: each pseudocode operation. */
  699: void eratosthenes(int numerator, int denominator, int chip,
  700:   struct statistics *s);
  701:
  702: int min(int a, int b);
  703:
  704: int main(int argc, char **argv){
  705:   struct chipStatistics summaryStatistics;
  706:
  707:   collectStatistics(&summaryStatistics, MAXNUMERATOR, MAXDENOMINATOR);
  708:
  709:   printf("Old chip (Euclid):\n");
  710:   printStatistics(&(summaryStatistics.oldChipEuclid));
  711:   printf("\n");
  712:   printf("New chip (Euclid)\n");
  713:   printStatistics(&(summaryStatistics.newChipEuclid));
  714:   printf("\n");
  715:   printf("Old chip (Sieve)\n");
  716:   printStatistics(&(summaryStatistics.oldChipSieve));
  717:   printf("\n");
  718:   printf("New chip (Sieve)\n");
  719:   printStatistics(&(summaryStatistics.newChipSieve));
  720:   printf("\n");
  721:
  722:   return 0;
  723: }
  724:
  725: void collectStatistics(struct chipStatistics *chipStats, int maxNumerat
or,
  726:   int maxDenominator){
  727:   int numerator, denominator;
  728:   /* Initialise all statistics */
  729:   initialiseStatistics(&(chipStats->oldChipEuclid));
  730:   initialiseStatistics(&(chipStats->newChipEuclid));
  731:   initialiseStatistics(&(chipStats->oldChipSieve));
  732:   initialiseStatistics(&(chipStats->newChipSieve));
  733:
  734:   for(numerator = 1; numerator <= maxNumerator; numerator++){
  735:     for(denominator = 1; denominator <= maxDenominator; denominator++){
  736:       /* Run algorithms for all combinations of numerator and denominat
or. */
  737:       euclid(numerator, denominator, OLDCHIP,
  738:         &(chipStats->oldChipEuclid));
  739:       euclid(numerator, denominator, NEWCHIP,
  740:         &(chipStats->newChipEuclid));
  741:       eratosthenes(numerator, denominator, OLDCHIP,
  742:         &(chipStats->oldChipSieve));
  743:       eratosthenes(numerator, denominator, NEWCHIP,
  744:         &(chipStats->newChipSieve));
  745:     }
  746:   }
```

```
747:   calculateAverage(&(chipStats->oldChipEuclid));
748:   calculateAverage(&(chipStats->newChipEuclid));
749:   calculateAverage(&(chipStats->oldChipSieve));
750:   calculateAverage(&(chipStats->newChipSieve));
751: }
752:
753: void calculateAverage(struct statistics *stats){
754:   stats->avgOperations = (double) stats->operations / stats->instances;
755: }
756:
757: void initialiseStatistics(struct statistics *stats){
758:   stats->operations = 0;
759:   stats->instances = 0;
760:   stats->minOperations = INT_MAX;
761:   stats->avgOperations = 0;
762:   stats->maxOperations = 0;
763: }
764:
765: /* actual euclid code not commented - just follows spec's pseudocode */
766: void
767: euclid(int numerator, int denominator, int chip, struct statistics *s){
768:     int b = numerator, a = denominator, temp;
769:     int counter=2; /* initial two assignments +2 */
770:
771:     counter++; /* counting for the final (failed) while +1 */
772:     while (b != 0){
773:         counter++; /* while loop +1 */
774:
775:         temp = b;
776:         counter++; /* assignment +1*/
777:
778:         b = a % b;
779:         counter++; /* assignment +1 */
780:         counter+=5; /* modulus +5 */
781:
782:         a = temp;
783:         counter++; /* assignment +1 */
784:     }
785:
786:     counter += 10; /* final 2 divisions before output +10 */
787:
788:     s->instances += 1;
789:     s->operations += counter;
790:
791:     if (s->maxOperations < counter){
792:         s->maxOperations = counter;
793:     }
794:     if (s->minOperations > counter){
795:         s->minOperations = counter;
796:     }
797: }
798:
799: /* actual eratosthenes code not commented - just follows spec's pseudoc
```

```
ode */
  800: void
  801: eratosthenes(int numerator, int denominator, int chip,
  802:   struct statistics *s){
  803:     int counter = 3;
  804:     int numCandidates = min(numerator, denominator), i, j;
  805:     int primes[numCandidates+1];
  806:
  807:     for (i=1; i<=numCandidates; i++){
  808:         primes[i] = 1;
  809:     }
  810:     counter++; /* assigning for the whole array +1 */
  811:
  812:     i = 1;
  813:     counter++; /* assigning +1 */
  814:
  815:     counter++; /* final (failed) while +1 */
  816:     while (i < numCandidates){
  817:         counter++; /* while loop +1 */
  818:
  819:         i++;
  820:         counter++; /* assigning after addition +1 */
  821:
  822:         counter++; /* evaluating an if - for below +1 */
  823:         if (primes[i]){
  824:             j=i+i;
  825:             if (chip == OLDCHIP) {
  826:                 /* part of line 13 so only needed for old chip */
  827:                 counter++; /* assigning the j +1 */
  828:             }
  829:
  830:             counter++; /* final (failed) while +1 AND ALSO the only cos
t
  831:             addition for whole line 13 for new chip */
  832:             while (j <= numCandidates){
  833:                 if (chip == OLDCHIP) {
  834:                     /* only for old chip! */
  835:                     counter++; /* evaluating one while loop above +1 */
  836:                     counter++; /* evaluating an if for below +1 */
  837:                     counter += 5; /* evaluating a division +5 */
  838:                     counter += 5; /* evaluating a mod +5 */
  839:                 }
  840:                 if (j/i > 1 && j%i == 0) {
  841:                     primes[j] = 0;
  842:                     if (chip == OLDCHIP){
  843:                         counter++; /* assigning +1 but only for old chi
p */
  844:                         counter++; /* j+=i assign below +1 but only for
 old
  845:                                       chip */
  846:                     }
  847:                 }
  848:                 j += i;
```

```
849:
850:             }
851:
852:             counter++; /* final (failed) while below +1 */
853:             counter+=5; /* final mod (failed) while below +5 */
854:             counter+=5; /* final mod (failed) while below +5 */
855:             while (numerator % i == 0 && denominator % i == 0){
856:                 counter++; /* evaluating while +1 */
857:                 counter+=5; /* evaluating a division +5 */
858:                 counter+=5; /* evaluating a mod +5 */
859:
860:                 numerator = numerator/i;
861:                 counter++; /* assigning +1 */
862:                 counter+=5; /* evaluating a division +5 */
863:
864:                 denominator = denominator/i;
865:                 counter++; /* assigning +1 */
866:                 counter+=5; /* evaluating a mod +5 */
867:             }
868:         }
869:     }
870:
871:     s->instances += 1;
872:     s->operations += counter;
873:
874:     if (s->maxOperations < counter){
875:         s->maxOperations = counter;
876:     }
877:     if (s->minOperations > counter){
878:         s->minOperations = counter;
879:     }
880: }
881:
882: /* helper function to return minimum of two numbers */
883: int
884: min(int a, int b){
885:     if (a > b){
886:         return b;
887:     } else {
888:         return a;
889:     }
890: }
891:
892: void printStatistics(struct statistics *stats){
893:   printf("Minimum operations: %d\n", stats->minOperations);
894:   printf("Average operations: %f\n", stats->avgOperations);
895:   printf("Maximum operations: %d\n", stats->maxOperations);
896: }
897:
898: ==> ./utils.c <==
899: /*
900: utils.c
901:
```

```
  902: Implementations for helper functions to do with reading and writing.
  903:
  904: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  905: */
  906: #include <stdio.h>
  907: #include <stdlib.h>
  908: #include <assert.h>
  909: #include "graph.h"
  910: #include "utils.h"
  911:
  912: struct graphProblem {
  913:   int antennaCost;
  914:   int numHouses;
  915:   int numConnections;
  916:   struct graph *graph;
  917: };
  918:
  919: struct graphProblem *readProblem(FILE *file){
  920:   int i;
  921:   int startHouse;
  922:   int endHouse;
  923:   int cost;
  924:   /* Allocate space for problem specification */
  925:   struct graphProblem *problem = (struct graphProblem *)
  926:     malloc(sizeof(struct graphProblem));
  927:   assert(problem);
  928:
  929:   /* First line of input is antenna cost. */
  930:   assert(scanf("%d", &(problem->antennaCost)) == 1);
  931:   /* Next line comprises number of houses and number of connections. */
  932:   assert(scanf("%d %d", &(problem->numHouses), &(problem->numConnection
s))
  933:     == 2);
  934:
  935:   /* Build graph number of houses + 1 because of datacentre. */
  936:   problem->graph = newGraph(problem->numHouses + 1);
  937:   /* Add all edges to graph. */
  938:   for(i = 0; i < problem->numConnections; i++){
  939:     assert(scanf("%d %d %d", &startHouse, &endHouse, &cost) == 3);
  940:     addEdge(problem->graph, startHouse, endHouse, cost);
  941:   }
  942:
  943:   return problem;
  944: }
  945:
  946: struct solution *findSolution(struct graphProblem *problem,
  947:   enum problemPart part){
  948:   return graphSolve(problem->graph, part, problem->antennaCost,
  949:     problem->numHouses);
  950: }
  951:
  952: void freeProblem(struct graphProblem *problem){
  953:   /* No need to free if no data allocated. */
```

```
  954:   if(! problem){
  955:     return;
  956:   }
  957:   freeGraph(problem->graph);
  958:   free(problem);
  959: }
  960:
  961: void freeSolution(struct solution *solution){
  962:   /* No need to free if no data allocated. */
  963:   if(! solution){
  964:     return;
  965:   }
  966:   free(solution);
  967: }
  968: ==> ./utils.h <==
  969: /*
  970: utils.h
  971:
  972: Visible structs and functions for helper functions to do with reading a
nd
  973: writing.
  974:
  975: Skeleton written by Grady Fitzpatrick for COMP20007 Assignment 1 2021
  976: */
  977: /* Because we use FILE in this file, we should include stdio.h here. */
  978: #include <stdio.h>
  979: /* Because we use struct graph in this file, we should include graph.h
here. */
  980: #include "graph.h"
  981: /* The problem specified. */
  982: struct graphProblem;
  983:
  984: /* Reads the data from the given file pointer and returns a pointer to
this
  985: information. */
  986: struct graphProblem *readProblem(FILE *file);
  987:
  988: /* Finds a solution for a given problem. */
  989: struct solution *findSolution(struct graphProblem *problem,
  990:   enum problemPart part);
  991:
  992: /* Frees all data used by problem. */
  993: void freeProblem(struct graphProblem *problem);
  994:
```