MATH182 HOMEWORK #5
DUE July 29, 2020

Exercise 1. *Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights with two distinguished vertices s and t. Describe a dynamic programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?*

Exercise 2. *A **palindrome** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).*

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

Solution. The following pseudocode[1] lays out a dynamic programming algorithm that finds the longest palindrome that is a subsequence of a given input string *str*:

```
FindPalindrome(str)
palindromes = empty matrix of dimension str.length × str.length
return FindPalindromeOfSubstring(str, 1, str.length + 1, palindromes)

FindPalindromeOfSubstring(str, s, e, palindromes)
// base case: substring str[s..e − 1] is empty
if e − s ≤ 0
    return empty string
// base case: substring str[s..e − 1] has length 1
if e − s == 1
    return str[s]
// base case: longest palindrome in substring str[s..e − 1] already found
if palindromes[s][e − 1] not empty
    return palindromes[s][e − 1]
// comparing first and last character to find maximum-length palindrome
if str[s] == str[e − 1]
    p = str[s] + FindPalindromeOfSubstring(str, s + 1, e − 1, palindromes) + str[e − 1]
else
    p1 = FindPalindromeOfSubstring(str, s, e − 1, palindromes)
    p2 = FindPalindromeOfSubstring(str, s + 1, e, palindromes)
    if p1.length > p2.length
        p = p1
    else p = p2
palindromes[s][e − 1] = p
return p
```

[1] C++ code in Appendix I

The running time of the algorithm is $O(n^2)$, since for each $s \in [1, str.\,length], e \in [2, str.\,length+1]$, we find the longest palindrome in $str[s\,..\,e-1]$ exactly once using only constant time operations such as comparing the longest palindrome subsequences in solved sub-problems.

Exercise 3. *Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.*

(1) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
(2) Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal optimal solution.
(3) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.
(4) Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

Exercise 4. *Let's consider a long, quiet conutry road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose the residents of these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.*

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Exercise 5. *Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (That is, if $V = \{v_1, \ldots, v_n\}$, then the degree of v_i should be exactly d_i.) G should not contain multiple edges between the same pair of nodes, or "loop" edges with both endpoints equal to the same node.*

Exercise 6. *A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.*

(1) Prove that in a breadth-first search of an undirected graph, the following properties hold:
   (a) There are no back edges and no forward edges.
   (b) For each tree edge (u, v), we have $v.\,d = u.\,d + 1$.
   (c) For each cross edge (u, v), we have $v.\,d = u.\,d$ or $v.\,d = u.\,d + 1$.
(2) Prove that in a breadth-first search of a directed graph, the following properties hold:
   (a) There are no forward edges.
   (b) For each tree edge (u, v), we have $v.\,d = u.\,d + 1$.
   (c) For each cross edge (u, v), we have $v.\,d \leq u.\,d + 1$.
   (d) For each back edge (u, c), we have $0 \leq v.\,d \leq u.\,d$.

Exercise 7. *An **Euler tour** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.*

(1) Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) *for each vertex $v \in V$.*
(2) Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists.

Exercise 8. *Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.*

We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be theset of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** *is a spanning tree T such that $w(T) = \min_{T'' \in \mathcal{T} \setminus \{T'\}} \{w(T'')\}$.*

(1) Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.
(2) Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T \setminus \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.
(3) Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $\max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $\max[u, v]$ for all $u, v \in V$.
(4) Give an efficient algorithm to compute the second-best minimum spanning tree of G.

Exercise 9 (Programming exercise)**.** *All square roots are periodic when written as continued fractions and can be written in the form:*

$$\sqrt{N} \;=\; a_0 + \cfrac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \cdots}}}$$

For example, let us consider $\sqrt{23}$:

$$\sqrt{23} \;=\; 4 + \sqrt{23} - 4 \;=\; 4 + \frac{1}{\frac{1}{\sqrt{23}-4}} \;=\; 4 + \frac{1}{1 + \frac{\sqrt{23}-3}{7}}$$

If we continue we would get the following expansion:

$$\sqrt{23} \;=\; 4 + \cfrac{1}{1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{8 + \cdots}}}}$$

The process can be summarized as follows:

$$\begin{aligned}
a_0 &= 4, & \frac{1}{\sqrt{23}-4} &= \frac{\sqrt{23}+4}{7} &&= 1 + \frac{\sqrt{23}-3}{7}\\
a_1 &= 1, & \frac{7}{\sqrt{23}-3} &= \frac{7(\sqrt{23}+3)}{14} &&= 3 + \frac{\sqrt{23}-3}{2}\\
a_2 &= 3, & \frac{2}{\sqrt{23}-3} &= \frac{2(\sqrt{23}+3)}{14} &&= 1 + \frac{\sqrt{23}-4}{7}\\
a_3 &= 1, & \frac{7}{\sqrt{23}-4} &= \frac{7(\sqrt{23}+4)}{7} &&= 8 + \sqrt{23} - 4\\
a_4 &= 8, & \frac{1}{\sqrt{23}-4} &= \frac{\sqrt{23}+4}{7} &&= 1 + \frac{\sqrt{23}-3}{7}\\
a_5 &= 1, & \frac{7}{\sqrt{23}-3} &= \frac{7(\sqrt{23}+3)}{14} &&= 3 + \frac{\sqrt{23}-3}{2}\\
a_6 &= 3, & \frac{2}{\sqrt{23}-3} &= \frac{2(\sqrt{23}+3)}{14} &&= 1 + \frac{\sqrt{23}-4}{7}\\
a_7 &= 1, & \frac{7}{\sqrt{23}-4} &= \frac{7(\sqrt{23}+4)}{7} &&= 8 + \sqrt{23} - 4
\end{aligned}$$

It can be seen that the sequence is repeating. For conciseness, we use the notation $\sqrt{23} = [4; (1, 3, 1, 8)]$, to indicate that the block $(1, 3, 1, 8)$ repeats indefinitely.

The first ten continued fraction representations of (irrational) square roots are:

$$
\begin{aligned}
\sqrt{2} &= [1;(2)] \quad \textit{period=1}\\
\sqrt{3} &= [1;(1,2)] \quad \textit{period=2}\\
\sqrt{5} &= [2;(4)] \quad \textit{period=1}\\
\sqrt{6} &= [2;(2,4)] \quad \textit{period=2}\\
\sqrt{7} &= [2;(1,1,1,4)] \quad \textit{period=4}\\
\sqrt{8} &= [2;(1,4)] \quad \textit{period=2}\\
\sqrt{10} &= [3;(6)] \quad \textit{period=1}\\
\sqrt{11} &= [3;(3,6)] \quad \textit{period=2}\\
\sqrt{12} &= [3;(2,6)] \quad \textit{period=2}\\
\sqrt{13} &= [3;(1,1,1,1,6)] \quad \textit{period=5}
\end{aligned}
$$

Exactly four continued fractions, for $N \leq 13$, have an odd period.

How many continued fractions for $N \leq 10000$ have an odd period?

Solution. oddPeriodSquareRoots takes input `bound` and returns[2] the number continued fractions of $\sqrt{N}$ that have odd period for $N \leq$ `bound`.

Calling oddPeriodSquareRoots(10000) returns <u>1322</u>.

```
#include<vector>
#include <math.h> // floor, sqrt
using namespace std;

vector<int> findContinuedFraction(int N);
    // defined in Appendix II

int oddPeriodSquareRoots(int bound) {
  // returns the number of continued fractions of root(n) that have an odd period for n <= bound

  int oddCount = 0;

  // finding continued fractions with odd period
  for (int N = 2; N <= bound; N++)
    if (findContinuedFraction(N).size() % 2 == 1)
      oddCount++;

  return oddCount;
}
```

Exercise 10 (Programming exercise)**.** *Consider quadratic Diophantine equations of the form:*

$$x^2 - Dy^2 = 1$$

For example, when $D = 13$, the minimal solution in x is $649^2 - 13 \times 180^2 = 1$.

It can be assumed that there are no solutions i positive integers when D is square.

[2]With the help of findContinuedFraction, a helper function defined in Appendix II.

By finding minimal solutions in x for $D = \{2, 3, 5, 6, 7\}$, we obtain the following:

$$\begin{aligned} 3^2 - 2 \times 2^2 &= 1 \\ 2^2 - 3 \times 1^2 &= 1 \\ 9^2 - 5 \times 4^2 &= 1 \\ 5^2 - 6 \times 2^2 &= 1 \\ 8^2 - 7 \times 3^2 &= 1 \end{aligned}$$

Hence, by considering minimal solutions in x for $D \leq 7$, the largest x is obtained when $D = 5$.

Find the value of $D \leq 1000$ in minimal solutions of x for which the largest value of x is obtained. (This link might be helpful: https://en.wikipedia.org/wiki/Pell%27s_equation)

Solution. DIOPHANTINEEQUATION takes input `bound` and returns, using FINDMINIMALSOLUTION, the value of $D \leq$ `bound` such that the minimal solution in x of $x^2 - Dy^2 = 1$ is largest.

Calling DIOPHANTINEEQUATION(1000) returns 661.[3]

```cpp
#include <vector>
#include <boost/multiprecision/cpp_int.hpp> // for large ints, from the non-standard boost library
using namespace boost::multiprecision;
using namespace std;

vector<int> findContinuedFraction(int N);
// defined in Appendix II
bool checkPerfectSquare(int N);
// defined in Appendix II
cpp_int findMinimalSolution(int D);


int diophantineEquation(int bound) {
  // return  non-perfect-square D in [2, bound] such that the minimal solution x for x^2 - Dy^2 = 1 is maximal

  // tracking max minimal solution found so far and the corresponding value of D
  cpp_int maxMinimalSolution = 0;
  int optimalD = 0;
  // iterating over [2, D]
  for (int D = 2; D <= bound; D++) {
    // filtering out perfect squares
    if (!checkPerfectSquare(D)) {
      // generating minimal solution
      cpp_int minimalSolution = findMinimalSolution(D);
      // updating maxMinimalSolution and optimalD as necessary
      if (minimalSolution > maxMinimalSolution) {
        maxMinimalSolution = minimalSolution;
        optimalD = D;
      }
    }
```

[3]This this is exactly half the answer for Exercise 9. That there might be a relationship doesn't seem entirely unreasonable, especially since continued fractions and the Diophantine Equation are obviously connected, but since the input for Exercises 9 and 10 are unrelated, I'm sceptical. It's probably just a fun coincidence.

```
  }
  // return optimal D
  return optimalD;
}

cpp_int findMinimalSolution(int D) {
  // finds minimal x such that x^2 - Dy^2 = 1 for some y given D

  vector<int> continuedFraction = findContinuedFraction(D);
  // constants in continued fraction of root D
  int period = continuedFraction.size();

  // initialising constants
  cpp_int Akm1 = 1;
  cpp_int Bkm1 = 0;
  cpp_int Ak = int(floor(pow(D, 0.5))); // floor of root D
  cpp_int Bk = 1;
  int k = 0;

  // while we don't have a solution to the Diophantine equation
  while (Ak * Ak - D * Bk * Bk != 1) {

    // generating next convergent Akp1/Bkp1
    // formula for next convergent verified from https://mathworld.wolfram.com/Convergent.html
    cpp_int Akp1 = continuedFraction[k % period] * Ak + Akm1;
    cpp_int Bkp1 = continuedFraction[k % period] * Bk + Bkm1;

    // updating variables
    Akm1 = Ak; Bkm1 = Bk;
    Ak = Akp1; Bk = Bkp1;
    k++;
  }

  return Ak;
}
```

Exercise 11 (Programming exercise)**.** *Consider the fraction $n/3$, where n and d are positive integers. If $n < d$ and $\gcd(n, d) = 1$, it is called a* ***reduced proper fraction****.*

If we list the set of reduced proper fractions for $d \leq 8$ in ascending order of size, we get:

$$1/8, 1/7, 1/6, 1/5, 1/4, 2/7, 1/3, 3/8, 2/5, 3/7, 1/2, 4/7, 3/5, 5/8, 2/3, 5/7, 3/4, 4/5, 5/6, 6/7, 7/8$$

It can be seen that $2/5$ is the fraction immediately to the left of $3/7$.

By listing the set of reduced proper fractions for $d \leq 1000000$ in ascending order of size, find the numerator of the fraction immediately to the left of $3/7$.

Solution. PREVIOUSREDUCEDFRACTION takes input `num`, `den` and `bound`, and returns the numerator of the reduced proper fraction immediately to the left of `num/den` when all the reduced proper fractions with denominator $d \leq$ `bound` are listed in ascending order.

Calling PREVIOUSREDUCEDFRACTION$(3, 7, 1000000)$ returns 428570.

```
#include <vector>
using namespace std;
```

```

void reduceToProper(int& n, int& d);
int euclid(int m, int n);

int previousReducedFraction(int num, int den, int bound) {
  // returns the numerator n of the reduced proper fraction n / d immediately to the left of num / den for n < d <= bound

  double upperBound = double(num) / den;  // upper bound for desired n / d
  // tracking the best reduced fraction and the corresponding numerator found so far
  double bestRatio = 0;
  int nOptimal = 0;

  // iterating over denominators d
  for (int d = 2; d <= bound; d++) {

    // a relatively tight, safe upper-bound for viable candidates
    int nStart = int(floor(d * upperBound));

    // decrementing nStart will we reach a viable candidate for n / d
    while (double(nStart) / d >= upperBound)
      nStart--;

    for (int n = nStart; n > 0; n--) {
      double ndRatio = double(n) / d;
      // terminating condition; if ndRatio < bestRatio, n/d is not a candidate and smaller n can also not be candidates
      if (ndRatio < bestRatio)
        break;

      // testing to see if we have a better candidate than our current best
      if (ndRatio > bestRatio) {
        // better candidate found -> find proper fraction
        int nReduced = n;
        int dReduced = d;
        reduceToProper(nReduced, dReduced);

        // update variables
        bestRatio = ndRatio;
        nOptimal = nReduced;
        // break, since any since any smaller n will give ndRatio < bestRatio
        break;
      }
    }
  }

  return nOptimal;
}

void reduceToProper(int& n, int& d) {
  // reduces n / d to a proper fraction

```

```
  int gcd = euclid(n, d);
  // iterate until n and d are co-prime
  while (gcd != 1) {
    // n and d are not co-prime; divide each by largest common factor
    n /= gcd; d /= gcd;
    gcd = euclid(n, d);
  }
}

int euclid(int a, int b) {
  // returns gcd(a, b)
  if (b == 0)
    return a;
  else return euclid(b, a % b);
}
```

Exercise 12 (Programming exercise)**.** *It is possible to write ten as the sum of primes in exactly five different ways:*

$$7+3$$
$$5+5$$
$$5+3+2$$
$$3+3+2+2$$
$$2+2+2+2+2$$

What is the first value which can be written as the sum of primes in over five thousand different ways?

Exercise 13 (Programming exercise)**.** *Do Project Euler Problem 83: Path sum: four ways. As a warmup, you might want to do problems 81 and 82 first.*

Solution. MINPATHSUMFOURWAYS returns the total sum along the minimal path[4] from the first element to the last element in a given matrix. The function finds the minimal path by using Djikstra's algorithm with an admissible heuristic for the total path length.

Calling MINPATHSUMFOURWAYS(“*problem83.txt*”, 80) returns <u>425185</u>.

```
#include <vector>
#include <queue>
#include <string>
#include "exercise13.h" // contains definitions for Node, Position
using namespace std;

struct Node; struct Position; struct NodeComparator;
// defined in header file, Appendix IV

vector<vector<int>> loadMatrix(string fileName, int dim);
// defined in Appendix III; loads matrix from text file
vector<vector<Node*>> generateNodeMatrix(vector<vector<int>>& numMatrix);
// generates and returns a matrix of nodes
bool withinBounds(Position& pos, int dim);
// returns whether a position is in bounds
```

[4]With help of helper function LOADMATRIX (code in Appendix III) and classes NODE, POSITION, and NODECOMPARATOR defined in header “exercise13.h” (code in Appendix IV)

```
int findHeuristic(Position& pos, int min, int dim);
// finds estimated "distance" from pos to last position

int minPathSumFourWays(string fileName, int dim) {
  // implements Djikstra's algorithm to find the minimal path top-left to bottom-right element

  // loading matrix
  vector<vector<int>> numMatrix = loadMatrix(fileName, dim);
  // matrix of nodes
  vector<vector<Node*>> nodeMatrix = generateNodeMatrix(numMatrix);

  // initialising algorithm
  priority_queue<Node*, vector<Node*>, NodeComparator> pQueue;  // priority queue for Djikstra' algorithm
  nodeMatrix[0][0]->shortestPath = nodeMatrix[0][0]->val;
  // our path starts with cost of first node
  pQueue.push(nodeMatrix[0][0]);

  // implementing Djikstra's algorithm based off of Computerphile video (https://www.youtube.com/watch?v=GazC3A4OQTE)
  while (!pQueue.empty()) {
    // extracting current node
    Node* node = pQueue.top(); pQueue.pop();

    // iterating over current nodes neighbours
    for (Node* adjNode : node->adjNodes) {
      // if adjacent node exists, updating adjacent node's shortest path as necessary
      if (adjNode != nullptr && node->shortestPath + adjNode->val < adjNode->shortestPath) {
        // update shortest path and shortestPath through
        adjNode->shortestPath = node->shortestPath + adjNode->val;
        adjNode->shortestPathThrough = node;
        // examine (or re-examine) neighbours of adjNode with new shortest path
        pQueue.push(adjNode);
      }
    }
  }

  // storing answer
  int answer = nodeMatrix[dim - 1][dim - 1]->shortestPath;

  // free memory taken by Nodes
  for (vector<Node*> row : nodeMatrix)
    for (Node* nodePtr : row)
      delete nodePtr;

  // return shortest path to last node
  return answer;
}

vector<vector<Node*>> generateNodeMatrix(vector<vector<int>>& numMatrix) {
```

```
// generate node matrix from number matrix

const int dim = numMatrix.size();

// finding minimum of numMatrix (for admissible heuristic)
int minNum = INT_MAX;
for (int rowNum = 0; rowNum < dim; rowNum++)
  for (int colNum = 0; colNum < dim; colNum++)
    if (numMatrix[rowNum][colNum] < minNum)
      minNum = numMatrix[rowNum][colNum];

vector<vector<Node*>> nodeMatrix(dim, vector<Node*>(dim, nullptr));

// creating node matrix (graph representation)
for (int rowNum = 0; rowNum < dim; rowNum++) {
  for (int colNum = 0; colNum < dim; colNum++) {

    // generating position and value of current node
    Position pos = Position(rowNum, colNum);
    int val = numMatrix[rowNum][colNum];

    // generating and storing node
    nodeMatrix[pos.row][pos.col] = new Node(pos, val, findHeuristic(pos, minNum, dim));
  }
}

// for each node, creating pointers to adjacent nodes
for (int rowNum = 0; rowNum < dim; rowNum++) {
  for (int colNum = 0; colNum < dim; colNum++) {

    Position pos = nodeMatrix[rowNum][colNum]->pos;
    vector<Node*> adjNodes(4, nullptr);

    // generating adjacent positions
    Position rightPos = Position(rowNum, colNum + 1);
    Position downPos = Position(rowNum + 1, colNum);
    Position leftPos = Position(rowNum, colNum - 1);;
    Position upPos = Position(rowNum - 1, colNum);

    // for each adjacent position, if position is in bounds,
      // insert a pointer to the corresponding node in adjNodePtrs
    if (withinBounds(rightPos, dim))
      adjNodes[RIGHT] = nodeMatrix[rightPos.row][rightPos.col];
    if (withinBounds(downPos, dim))
      adjNodes[DOWN] = nodeMatrix[downPos.row][downPos.col];
    if (withinBounds(leftPos, dim))
      adjNodes[LEFT] = nodeMatrix[leftPos.row][leftPos.col];
    if (withinBounds(upPos, dim))
      adjNodes[UP] = nodeMatrix[upPos.row][upPos.col];

    // storing adjNodePtrs
    nodeMatrix[pos.row][pos.col]->adjNodes = adjNodes;
```

```
    }
  }

  return nodeMatrix;
}

bool withinBounds(Position& pos, int dim) {
  // checks whether a given position is within bounds
  return (pos.row >= 0 && pos.row < dim && pos.col >= 0 && pos.col < dim);
}

int findHeuristic(Position& pos, int min, int dim) {
  // note: for the heuristic to be admissible, we want to ensure we never overestimate the actual cost of getting
    // from a node to the last node. We therefore define the heuristic of a given node to be the total cost of getting from
    // the node the last node assuming the shortest-length path is followed and each element is the smallest in the matrix
  return min * ((dim - pos.row - 1) + (dim - pos.col - 1));
}
```

APPENDIX

I. C++ code for Exercise 2:

```cpp
#include <string>
#include <vector>
using namespace std;

string findPalindromeOfSubstring(string str, int s, int e,
    vector<vector<string*>>& palindromes);

string findPalindrome(string str) {
  vector<vector<string*>> palindromes
      (str.length(), vector<string*>(str.length(), nullptr));
    // palindromes[s][e - 1] stores the longest palindrome in str[s : e - 1] (inclusive); stores nullptr if palindrome not yet found

  // generating answer
  string ans = findPalindromeOfSubstring(str, 0, str.length(), palindromes);

  // freeing memory
  for (vector<string*> row : palindromes)
    for (string* p : row)
      if (p != nullptr)
        delete p;

  return ans;
}

string findPalindromeOfSubstring(string str, int s, int e,
    vector<vector<string*>>& palindromes) {
  // base case: empty string
  if (e - s == 0)
    return "";
  // base case: string of length
  else if (e - s == 1)
    return string(1, str[s]);
  // base case: longest palindrome already found
  else if (palindromes[s][e - 1] != nullptr)
    return *palindromes[s][e - 1];

  // variable to store answer
  string p;

  // comparing first and last character
  // first and last character match
  if (str[s] == str[e - 1]) {
    p = str[s]
        + findPalindromeOfSubstring(str, s + 1, e - 1, palindromes)
        + str[e - 1];
  }
  // first and last characters don't match
  else {
```

```
    // solving subproblems
    string p1 = findPalindromeOfSubstring(str, s + 1, e, palindromes);
    string p2 = findPalindromeOfSubstring(str, s, e - 1, palindromes);
    // finding optimal solution among subproblems
    if (p1.length() >= p2.length())
      p = p1;
    else p = p2;
  }

  // storing found palindrome for future reference
  palindromes[s][e - 1] = new string(p);
  // returning answer
  return p;
}
```

II. C++ code for helper functions findContinuedFraction and checkPerfectSquare used in Exercises 9 and 10:

```
#include <vector>
using namespace std;

// used for exercise 9, 10
bool checkPerfectSquare(int N);
vector<int> findContinuedFraction(int N);

bool checkPerfectSquare(int N) {
  // checks whether a number is a perfect square
  if (pow(int(sqrt(N)), 2) == N)
    return true;
  else return false;
}

vector<int> findContinuedFraction(int N) {
  // returns the period of the continued fraction for root n

  const int aStart = int(floor(sqrt(N))); // termination check: when our next expression is root(n) - aStart, we can terminate

  // checking if N is a perfect square
  if (aStart * aStart == N)
    return vector<int> {};  // no continued fraction

  // sequence of a's (until sequence starts repeating
  vector<int> fractionConstants = { };

  int nK = 1;
  int dK = -aStart; // in reality, denominator is root(n) - denominatorTerm

  do {
    // rationalising numerator / (root(n) - denominatorTerm)
    int dKp1 = -dK; // (tentative) next denominator
    int nKp1 = (N - int(pow(dK, 2))) / nK;  // next numerator

    int aKp1 = 0;
```

```

    // transforming dKp1 to appropriate form
    while (N - int(pow(dKp1 - nKp1, 2)) > 0) {
      dKp1 -= nKp1;
      aKp1++;
    }

    // updating nK, dK, fractionConstants
    nK = nKp1; dK = dKp1;
    fractionConstants.push_back(aKp1);

  } while (nK != 1 || dK != -aStart);
  // if while condition is met, we're back to the first iteration and constants will repeat

  // returning continued fraction constants
  return fractionConstants;
}
```

III. C++ code for LOADMATRIX, a helper function for Exercise 13:

```
#include <vector>
#include <fstream>  // defines ifstream
#include <string>
#include <iostream>
using namespace std;

// used for exercise 13
vector<vector<int>> loadMatrix(string fileName, int dim);

vector<vector<int>> loadMatrix(string fileName, int dim) {
  // loads matrix defined in input file with address "fileName" (rows separated by '\n', ints separated by ", ")
    // into square, 2D vector matrix

  vector<vector<int>> matrix;

  // loading file into inputFile
  ifstream inputFile(fileName);

  // file could not be found
  if (!inputFile) {
    cerr << "File could not be found." << endl;
    exit(1);
  }

  // iterate over rows; with each iteration, we generate a row vector
  for (int rowNum = 0; rowNum < dim; rowNum++) {
    vector<int> row;
    // iterating over each element in row (except last)
    for (int colNum = 0; colNum < dim; colNum++) {
      int rowElem;
      inputFile >> rowElem; // extracting row element
      if (colNum < dim - 1)
```

```
        inputFile.ignore(10000, ','); // skip comma before next element
          // (only exists if not last element)
      row.push_back(rowElem); // inserting row element into row
    }
    matrix.push_back(row);  // inserting row
  }

  return matrix;
}
```

IV. C++ code for classes Node, Position, and NodeComparator, helper classes for Exercise 13:

```
#pragma once

#include <vector>
using namespace std;

enum direction { RIGHT, DOWN, LEFT, UP };

struct Position {
  // class defined to keep track of positions travelled in paths

  // private data members
  int row;
  int col;

  // constructor
  Position(int rowIndex = -1, int colIndex = -1)
    : row(rowIndex), col(colIndex) {}
};

struct Node {
  // const data members
  const Position pos; // position of node
  const int val;  // value of node
  vector<Node*> adjNodes = vector<Node*>(4, nullptr); // pointers to adjacent nodes
    // assumed no adjacent nodes by default
    // adjNodePtrs should also be const in theory, but coded non-const to make generateNodeMatrix simpler

  // non-const data members for use in Djikstra's algorithm
  int shortestPath = INT_MAX; // sentinel for infinity
  Node* shortestPathThrough = nullptr;  // node through which shortest path is attained
    // not strictly necessary for the sum of the shortest path, but can be used to find the shortest path
  int heuristic;  // heuristic (representative of estimated "distance" from node to ending node

  // constructor (if called with no parameters, creates dummy nodes)
  Node(Position pos = Position(-1, -1), int val = -1, int heuristic = 0)
    : pos(pos), val(val), heuristic(heuristic) {}
```

```cpp
};

struct NodeComparator {
  // comparator class for min-heap priority queue
  bool operator() (const Node* node1, const Node* node2) {
    return node1->val + node1->heuristic < node2->val + node2->heuristic;
    // returns true if the path through node1 has total estimated cost less than the path through node2
  }
};
```