Last name(s) **Name** **ID**

Final EDA Exam **Length: 3 hours** **12/01/2017**

- *The exam has 4 sheets, 8 sides and 4 problems.*
- *Write your full name and ID on every sheet.*
- *Write your answers within the reserved space in the exam sheet.*

Problem 1 (2 pts.)

Fill the following gaps as precisely as possible:

(a) (0.25 pts.) A graph with n vertices has $O(\quad)$ edges.

(b) (0.25 pts.) A connected graph with n vertices has $\Omega(\quad)$ edges.

(c) (0.25 pts.) A complete graph with n vertices has $\Omega(\quad)$ edges.

(d) (0.25 pts.) A min-heap with n vertices has $\Theta(\quad)$ leaves.

(e) (0.25 pts.) A binary search tree with n vertices has height $\Omega(\quad)$.

(f) (0.25 pts.) A binary search tree with n vertices has height $O(\quad)$.

(g) (0.25 pts.) An AVL tree with n vertices has height $\Omega(\quad)$.

(h) (0.25 pts.) An AVL tree with n vertices has height $O(\quad)$.

This side would be intentionally blank if it were not for this note.

Last name(s) **Name** **ID**

Problem 2 **(3 pts.)**

Let $G = (V, E)$ be a directed graph with weights $\omega : E \to \mathbb{R}$. We want to solve the problem of, given a vertex $s \in V$, to compute the distance from s to all vertices of the graph.

Recall that:

- a *path* is a sequence of vertices that are connected consecutively by arcs; that is to say, $(u_0, u_1, \ldots, u_k)$ such that for all $1 \leq i \leq k$, we have $(u_{i-1}, u_i) \in E$.
- the *weight* of a path is the sum of the weights of its arcs:

$$\omega(u_0, u_1, \ldots, u_k) = \sum_{i=1}^{k} \omega(u_{i-1}, u_i)\,.$$

- the *distance* of a vertex u to a vertex v is the weight of the path (called *minimum path*) with minimum weight among those leaving from u and arriving at v, if it exists.

(a) (0.2 pts.) Let us assume that for any edge $e \in E$, we have $\omega(e) = 1$; that is to say, all weights are 1. Which algorithm can we use to solve efficiently the problem of the distances?

(b) (0.2 pts.) Let us assume that for any edge $e \in E$, we have $\omega(e) \geq 0$; that is to say, all weights are non-negative. Which algorithm can we use to solve efficiently the problem of the distances?

(c) (0.2 pts.) Which algorithm can we use to solve efficiently the problem of the distances, if some weights can be negative?

(d) (0.4 pts.) Which condition over the cycles of the graph must be satisfied so that, for all pairs of vertices $u, v \in V$, there exists a minimum path from u to v?

(e) (1 pt.) A function $\pi : V \to \mathbb{R}$ is a *potential* of the graph if it satisfies that, for any edge $(u, v) \in E$, we have $\pi(u) - \pi(v) \leq \omega(u, v)$. Moreover, the *reduced weights* ω_π are defined as $\omega_\pi(u, v) = \omega(u, v) - \pi(u) + \pi(v)$ for any $(u, v) \in E$.

Prove that if c is a path from u to v then $\omega_\pi(c) = \omega(c) - \pi(u) + \pi(v)$.

(f) (1 pt.) Suppose that the graph has a potential π. Then, it can be proved that for all pairs of vertices $u, v \in V$ there is a minimum path with weights ω from u to v. Assuming this fact, explain how to use the potential π to compute the distance from a given vertex s to all vertices of the graph with an alternative algorithm to that of part (c) when the weights can be negative.

Last name(s) **Name** **ID**

Problem 3 **(2 pts.)**

We define a type *matrix* to represent square matrices of real numbers. Consider the following program:

```
typedef vector<vector<double>> matrix;

matrix aux(const matrix& A, const matrix& B) {
  int n = A.size ();
  matrix C(n, vector<double>(n, 0));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      for (int k = 0; k < n; ++k)
        C[i][j] += A[i][k] * B[k][j];
  return C;
}

matrix mystery(const matrix& M) {
  int n = M.size ();
  matrix P(M), Q(M);
  for (int i = 1; i < n; ++i) {
    P = aux(P, M);
    Q = aux(Q, P);
  }
  return Q;
}
```

(a) (0.5 pts.) What does the function *matrix mystery*(**const** *matrix*& M) compute in terms of the matrix M?

(b) (0.5 pts.) If M is an $n \times n$ matrix, what is the cost in the worst case of the function *matrix mystery*(**const** *matrix*& M) as a function of n? Justify your answer.

(c) (1 pt.) Implement in C++ a function that computes the same as the function *matrix mystery*(**const** *matrix*& M) and which takes $\Theta(n^3 \log n)$ time in the worst case. Also implement the auxiliary functions you use, except for *aux* and the functions of the standard library of C++. Justify that the cost of your function is as required.

Last name(s) **Name** **ID**

Problem 4 **(3 pts.)**

The problem of HAMILTONIAN GRAPH consists in, given an (undirected) graph, to decide whether it is Hamiltonian, that is to say, whether there exists a cycle that visits all its vertices exactly once. It is well-known that HAMILTONIAN GRAPH is an NP-complete problem.

It is also known that, from the proof that a problem belongs to class NP, one can derive a brute force algorithm that solves it. The following function **bool** *ham*(**const** *vector*<*vector*<**int**>>& *G*) which, given a graph *G* represented with adjacency lists, returns if *G* is Hamiltonian, implements this algorithm in the case of HAMILTONIAN GRAPH.

```
bool ham_rec(const vector<vector<int>>& G, int k, vector<int>& p) {
  int n = G.size();
  if (k == n) {
    vector<bool> mkd(n, false);
    for (int u : p) {
      if (mkd[u]) return false;
      mkd[u] = true;
    }
    for (int k = 0; k < n; ++k) {
      int u = p[k];
      int v;
      if (k < n-1) v = p[k+1];
      else         v = p[0];
      if (find(G[u].begin(), G[u].end(), v) == G[u].end()) return false;
    }
    return true;
  }
  for (int v = 0; v < n; ++v) {
    p[k] = v;
    if (ham_rec(G, k+1, p)) return true;
  }
  return false;
}

bool ham(const vector<vector<int>>& G) {
  int n = G.size();
  vector<int> p(n);
  return ham_rec(G, 0, p);
}
```

Note: The function of the STL library

```
Iterator find(Iterator first, Iterator last, int val);
```

returns an iterator to the first element in the range $[first, last)$ that compares equal to *val*. If no such element exists, the function returns last.

(a) (0.5 pts.) Identify the variable in the previous program which represents the witness when the function *ham* returns **true**.

(b) (0.5 pts.) Identify the code corresponding to the verifier in the previous program. To that end, use the line numbers on the left margin.

(c) (1 pt.) Fill the following gaps so that the function *ham2* computes the same as the function *ham*, but more efficiently.

```
bool ham2_rec(const vector<vector<int>>& G, int k, int u, vector<int>& next) {
  int n = G.size();
  if ( [    ] == n)
    return find(G[u].begin(), G[u].end(), [    ] ) ≠ G[u].end();

  for (int v : G[u])
    if (next[v] == [    ] ) {
      next[u] = [    ] ;
      if (ham2_rec(G, k+1, v, next)) return true;
      next[u] = −1;
    }
  return false;
}

bool ham2(const vector<vector<int>>& G) {
  int n = G.size();
  vector<int> next(n, −1);
  return ham2_rec(G, [    ] , 0, next);
}
```

(d) (0.5 pts.) Suppose that the adjacency lists of the representation of G are sorted (for example, increasingly). Explain how to use this to make the function of part (c) more efficient.

(e) (0.5 pts.) Suppose that G is a **disconnected** graph. Explain how to use this to make the function of part (c) more efficient.