Answers and comments to exam on 2017-06-02 in DA3018

1. (a) I choose to store the 12-digit identity number as a 12-element byte array, which needs 12 bytes. There is no need to store the hyphen. 1 MB equals 10^6 bytes, so $10^6/1283\ 333$ identity numbers fits in that memory.

   - It is OK to say that 1 MB equals $1024^2 = 2^{20}$ bytes too, which some prefer, but these days that is formally written as 1 MiB.
   - One can consider storing parts of the identity number as integers too, but I am guessing that is not convenient in general. If year, month, day, and the last four digits were stored as four separate integers, we would need $4 \times 4 = 16$ if "typical" integer type is used.
   - It is natural to think of using a string for representing a personal identity number. We have not talked about string representation in the course, and their representation varies between programming languages, so you have some freedom in your assumptions. In old languages like C, strings are variable length, encoding characters in the old ASCII standard in which each character is a byte, and the string is terminated by a zero-byte. Thus, a string of c characters take $c + 1$ bytes. In Java, a character is two or four bytes because Java is supposed to handle strings from many languages and uses the Unicode standard for character representation. Only the two-byte representation is needed for digits, so 12 digits would then need 24 bytes.

   (b) A single-linked list is a list of nodes containing a data element and a memory address (pointer/reference, depending on your programming language) for the next node. At the last node the adress is 0 ("null" in Java) to signify list termination. In Java, the list node would contain a reference to the data element. In some languages one could store the data "natively" in the node.

   I assume we work on a modern 64-bit computer, so the address part of each node is 8 bytes. It is fine to assume a 32-bit computer too, in which case you need 4 bytes.

   Going with a Java-native estimate, the node contains two 8-byte references for each id, so 16 bytes and an overhead of $(8 + 8)/12 = 133\ \%$.

2. (a) Python-inspired pseudocode:

```
def xor(A, B):
    k = size(A)
    X = new Array(k) of bytes
    for i in [1, k]:
        X[i] = (!A[i] && B[i]) || (A[i] && !B[i])
    return X
```

   (b) We will use a version of binary search in A, so we will keep track of *left* and *right* of a region in A that we are zooming in on. The

left index is the first element of the region, and the right index is the non-included region-ending element. If we end up with an empty region, *left* == *right*, then we give up and return null.

```
def find(A, x):
    return find(A, 0, size(A), x)

def find(A, left, right, x):
    if left == right
        return null
    else:
        mid = (left + right) // 2
        L = A[mid][0]
        R = A[mid][1]
        if L <= x and x <= R:
            return L, R
        else:
            if x < L:
                return find(A, left, mid, x)
            elif L < x:
                return find(A, mid, right, x)
            else:
                return null
```

The algorithm will terminate in time $O(\lg n)$, because the region of A that we consider will be split in two in each recursive call. If there are n elements in A, then subarray will be reduced to a single element in $O(\lg n)$ recursions.

The base case is trivially correct. If *left* and *right* are identical, then there is not interval to find x in.

If it is not an empty region of A we are looking at, we start by considering the middle element. If x is in its interval, then it is returned. Otherwise, we check what side of L that x is found on.

Suppose that there is an interval $x \in [\hat{L}, \hat{R}]$ in A. If $x < L$, then $\hat{L} < L$, so recursing on items *left* to *mid* is correct.

Consider the case $L < x$. If $\hat{L} < L$ and $x \notin [L, R]$ then $R' < R$, so it should hold that $x \in [L, R]$ as well and we would have returned $[L, R]$ as a solution. If instead $L < \hat{L}$, then it is correct to recurse on *mid* to *right* in A.

(c) Here is pseudo code for a suggested solution.

```
def eval(expr):
    operator, args = expr  # Extract the parts from
         the node with patterns matching
    if operator == C:  # Base case
        return args # True/False
    else:
        operands = map(eval, args)  # Call eval
            recursively on each operand
        if operator == AND:
            return all(operands)
        else:
            return any(operands)
```

The function is correct, because we have a base case for constants: if the expression is a constant, then the contant value is return. If the operator is AND or OR, then all operands are evaluated first, and in the case of an AND operator we make sure that all operands evaluated to True using **`all`**, whereas for OR we test whether one operand evaluated to True using **`any`**.

In this solution, I have used the **`all`** and **`any`** functions from Python, that can be written as follows (in pseudo code):

```
def all(operands):
    res = True
    for x in operands:
        if not x:
            return False
    return True
```

The **`all`** function loops through all operands. If one is False, then the loop is escaped early with **`return`** `False`, otherwise the loop will finish and True is returned.

```
def any(operands):
    for x in operands:
        if x:
            return True
    return False
```

The **`any`** function is similar, but returns early if True is found.

(d) Create a graph $G = (V, E)$ with a vertex for each tile, and an edge when two adjacent tiles have connecting roads. The map in the figure of the exam has four vertices and three edges.

Let v_s and v_e be the vertices representing start and end tiles. We want to find the shortest path p between v_s and v_e. The way we define the graph, the route distance will equal the number of edges in p, i.e., the length of the path. The shortest path from a start vertex to every other vertex can be computed using breadth-first search.

Consider this algorithm:

```
def distance(G, start):
   # Initialize
   for v in V(G):
      v.visited = False

   # Start computing distances
   Q = new Queue()
   Q.enqueue(start)
   start.visited = True
   d[start] = 0    # Distance to itself is 0
   while not Q.empty():
      v = Q.dequeue()
      for u in neighbors(v):
         if not u.visited:
            Q.enqueue(u)
            u.visited = True
            d[u] = d[v] + 1
```

The length of the shortest route is found by $L[v_e]$, after $L = \text{distance}(G, v_s)$.

In the algorithm, we will set the distance to v_s's neighbors to 1, and then (using that the neighbors of v_s has been put in the queue) set the distance to the neighbors' neighbors to 2, etc.

We can use induction to argue the correctness. As a base case, the distance is trivially correct for v_s (= `start` in the pseudo code).

As an inductive step, assume that $(v, u) \in E$ and distances have been correctly computed for v and all other vertices w with $d(w) \leq d[v]$. There are two cases.

- If the shortest path from v_s to u is through v, then the distance to u is computed corrrectly, because u cannot have been put in the queue and been visited before v, and the algorithm sets $d[u] = d[v] + 1$.
- If there is a shorter path to u, not going through v, then there must be another vertex w with $(w, v) \in E$, and w must have been visited before v. However, this breaks the induction hypothesis, since u would already have been marked as visited.

Because the algorithm is an adaption of BFS, adding only $O(1)$ operations in each step of the algorithm, we stay within the $O(|V|+|E|)$ time complexity guarantee of BFS. Our map have n tiles, so $|V| = n$, and there are at most 4 roads from each tile, so $|E| \leq 4n/2 = 2n$. Therefore, the time complexity of our algorithm is $O(n+4n) = O(n)$.

(e) Here is a possible problem formulation.

The grade partitioning problem:

- **Input:** An integer k for the number of forms (groups) to be created, an integer m the maximum number of children in each form, a set of children C, and for each child $c \in C$ there is a subset F_c of friends, $F_c \subset C$ that c would like to be together with the next year.
- **Output:** A partitioning $P_1, P_2, \ldots, P_k$ such that $\bigcup_{i=1}^{k} P_i = C$, and the number of pairs $c_1, c_2 \in C$ such that $c_1 \in P_i$ and $c_2 \in P_j$, $i \neq j$, is minimized.

For full points it should be clear what the input and the output is.