Exercise 1

a)
The problem presented maps to a simple 0-1 knapsacking problem. Therefore, we find a solution using the following intuition: we arbitrarily arrange the projects into an array. Then, we iterate over all projects and for each project p we ask the following question: does including this project bring a benefit or not?
We answer the question by comparing two cases:

I) we include the project. Then, we have obtained a reduced version of the problem with a budget $B' = B - c(p)$, a set of projects $P' = P - p$. Assuming we know the optimal utility for our reduced problem u'_{max}, the maximum achievable utility in this scenario is $u_I = u'_{max} + u(p)$.

II) we do not include the project. Then, we consider a similarly reduced version of our problem - with an untouched budget, but a smaller set of projects $P' = P - p$. Here, the maximum achievable utility is $u_{II} = u''_{max}$, with u''_{max} being the optimal utility for the reduced problem.

By comparing the two utilities, we can choose whether or not to include a project. Iterating this process reduces the problem down to a trivial case. By first building a table from small to large, and then iterating back down, we can achieve efficient computation.

In short, consider the pseudocode (on page 2)

The proof for correctness more or less follows this intuition.
Considering any problem where we have a budget B and an array of projects P, with cost and utility functions.

Iff the optimal values for t[p-1, b] and t[p-1, b-c[p]] were correctly calculated, then max(t[p-1, b], u(p) + t[p-1, b-c[p]]) is also optimal, as a project can only be taken or not taken.

Since the values for p = 0 and p = 1 are trivially optimal, the other values are therefore also optimal. Therefore, the algorithm returns the correct subset of projects.

Furthermore, given the construction of the table, calculated subsets must always be budget-respecting.

```
Input:
array of projects P
one-indexed array of costs c
one-indexed array of utilities u
integer budget B

// build DP table
t <- 0-initialised 2d array with |P|+1 rows, B+1 columns
for p in [1, |P|]:
    for b in [1, B]:
        if c[p] > b:
            // project does not fit into budget
            t[p, b] <- t[p-1, b]
        else:
            // either take or do not take project
            t[p, b] <- max(t[p-1, b], u(p) + t[p-1, b-c[p]])

// extract set of projects
p <- |P|
b <- B
S <- set of projects
while p * b != 0:
    if t[p, b] > t[p-1, b]:
        S <- S + {P[p]}
        p <- p - 1
        b <- b - c[p]
    else:
        p <- p - 1
return S
```

b)
In order to maximize the minimal distance, we formulate a decision problem: given a finite set of points $U \subset [0, 1]$, an integer $k \in \mathbb{N}$ and a distance $d \in \mathbb{R}$, can k points be selected such that the minimal distance between points is larger than d?

We solve this decision problem by defining an answer set S. The smallest point in U is included in S. Then, we iteratively pick the smallest point at least d away from the largest point in S, until no such point exists. If $|S|$ ¿ k, we return true - if not, we don't. For convenience, we also return the set of picked points.

We then perform a binary search over distances d, beginning with lower and upper bounds 0 and 1 and initial distance 0.5 to find the largest possible d where the decision problem returns true. Then, the set of points with the largest minimum distance is obtained by running the decision problem algorithm for said largest distance d_{max} again, and we compute its minimum distance to obtain the maxmin distance.

As d is a real number, we need to define an early termination criterion: if running our decision problem algorithm for both the lower bound l and upper bound u of the binary search results in the same set, further searching is unnecessary and we terminate.

Consider in this regard the pseudocode on the next page.

Algorithm correctness is shown as such:

First, the solution to the decision problem must be correct. Given any subset S of U with at least k elements and a minimum distance $\geq d$, the smallest point in S either is, or may be exchanged for the smallest point in U without reducing the minimum distance. Therefore, including the smallest point is correct. Then, the rest of the solution is also trivially correct.

Assume that the set of points returned by the algorithm is incorrect, there is some actual maximal minimum distance d_{corr} and we consider two cases:

Case I:
The distance d_{corr} lay within the bounds of our final iteration of binary search. Therefore, the decision problem algorithm for the upper bound would have returned a different set of points than the algorithm for the lower bound. Therefore, we would not have terminated when we did, and this case leads to a contradiction.

Case II:
The distance d_{corr} lay outside of the bounds of our final iteration of binary search. As all possible distances are covered by our binary search, we must at some point have moved a bound past d_{corr}. If the bound moved past d_{corr} was the upper bound, this implies the decision problem returned false for some d ¡ d_{corr} and vice versa for the lower bound. As this is impossible, this case also leads to a contradiction.

Therefore, the assumption inherently leads to a contradiction.

```
Input:
set of reals U
integer k

function distancedPoints(array U, int d):
    s <- set of reals
    s <- s + {U[0]}
    last <- U[0]
    for r in U:
        if r >= last + d:
            last <- r
            s <- s + {r}
    return s
l <- 0
u <- 1
U' <- sort(array(U))
while True:
    d <- (l+u) / 2
    p <- distancedPoints(U', d)
    if |p| >= k:
        l <- d
    else:
        u <- d
    if distancedPoints(U', l) == distancedPoints(U', u):
        break
p <- distancedPoints(U', d)
d <- diversity(p)
return d
```

Exercise 2

a)
Given an instance of VC, consisting of a Graph $G = (V, E)$ and a positive integer $k \in \mathbb{N}$, we construct an instance of FVS as follows:

First, we construct a set of nodes consisting of the nodes in V with additional nodes for edges in E: $V' = V \cup V_E$, with $V_E = \{v_{x,y} for (x, y) \in E\}$. Then, we construct a set of arcs A as follows for every edge in E, we make back-and-forth arcs in A. Furthermore, we additionally construct edges for our newly constructed edge-nodes, i.e. $A = \{(x, y), (y, x), (x, v_{x,y}), (v_{x,y}, y) \mid (x, y) \in E\}$. Finally, we construct a directed Graph $G' = (V', A)$

Now, we propose that there is a solution to VC for k on G iff there is a solution to FVS for k on G'.

$\rightarrow$
Given a subset S of V that solves VC for k on G, we solve FVS on G' as follows:
The set S represents a solution to FVS on G'. As every cycle includes at least one arc, and we didn't construct any reflexive arcs (x,x), a cycle therefore includes at least two vertices. If these two vertices are within V, the arc was constructed from an edge in E. Therefore, the VC properties of S imply that at least one of the vertices is in S. We didn't construct any arcs between only vertices outside of V. If one vertex is within V and the other is not, these vertexes must be a pair $(x, v_{x,y})$ or $(v_{x,y}, y)$. Therefore, the cycle in question must contain the path $(x, v_{x,y}, y)$, where once again either x or y are within S as it is a VC set. Therefore, S solves FVS on G' with $|S| = k$.

$\leftarrow$
Given a subset S of V' that solves FVS for k on G', we solve VC on G as follows:
For every edge $(x, y) \in E$, there exists a cycle $(x, v_{x,y}, y)$ in G'. As S satisfies FVS, we therefore know that $\{x, v_{x,y}, y\} \cap S \neq \emptyset$. We define a subset $S' \subseteq G = \{v \mid v \in S \cap G\} \cup \{x \mid v_{x,y} \in S - G\}$, that is - we keep vertices originally present in G, and replace constructed vertices with neighbouring original ones. As such, $S' \subseteq G$. As $\forall (x, y) \in E : \{x, v_{x,y}, y\} \cap S \neq \emptyset$, therefore $\forall (x, y) \in E : \{x, y\} \cap S' \neq \emptyset$, i.e. S' solves VC on G. Since $|S| = k$, $|S'| \leq k$. q.e.d.

b)
We consider a 3SAT instance consisting of clauses C over variables U, with $|c| = 3 \forall c \in C$.
For this reduction, we build some circuits:
For a given 3SAT variable x, we construct a variable circuit as follows: we construct two nodes v_x and $v_{\neg x}$, connected by an edge $(v_x, v_{\neg x})$.

For a given 3SAT clause $a \vee b \vee c$, where abc correspond either to a variable or a negated variable, we construct a clause circuit as follows: we construct three nodes referred to within the circuit as v_a, v_b, v_c, and connect them by edges $(v_a, v_b), (v_a, v_c), (v_b, v_c)$.

Finally, we construct edges connecting variable and clause circuits. Each node within a variable circuit corresponding to truth of a variable is connected to the corresponding node on all clause circuits. That is, a variable circuit node $v_{\neg x}$ is connected by an edge to all clause circuit nodes $v_{\neg x}$.

Now to prove that there is a solution to our 3SAT problem iff there is a solution to VC on our constructed graph with $k = |U| + 2 \cdot |C|$:
$\rightarrow$
Given a truth assignment to U such that all clauses c evaluate to true, we construct a solution to VC as follows:
According to the truth assignment, we pick one node per variable circuit, that is if variable x is true, we pick v_x and if false, we pick $v_{\neg x}$.

Then, we pick two variables per clause circuit. As our truth assignment satisfies 3SAT, at least one part of each clause is true, and therefore the edge connecting at least one node in the clause circuit to a node in a variable circuit is covered by the node in the variable circuit. We consider such an edge, then pick the two nodes not related to said edge.

Because we picked one node per variable circuit and two nodes per clause circuit, all circuit-internal edges are covered. As shown above, all inter-circuit edges are covered by either a node from a variable or a node from a clause circuit.

Therefore, we have picked k nodes such that all edges are covered, and thereby solved VC on the constructed graph.

$\leftarrow$
Given set of nodes S fulfilling VC with $k = |U| + 2 \cdot |C|$ on the constructed graph. We now prove that there is a truth assignment to U satisfying all clauses C and thereby 3SAT.

Considering the internal edges within the circuits, S must include exactly two nodes per clause circuit and one node per variable circuit. Therefore, there must be one node within each clause circuit not in S, and furthermore an edge connecting this node to a node within a variable circuit. As this edge is covered, the related variable circuit node is within S. Therefore, assigning truth to all variable circuit nodes within S provides a truth assignment satisfying all clauses C, thereby satisfying 3SAT. q.e.d.

Exercise 3

a)
The corresponding decision problem is called maximum set coverage. It reads as follows: Given a set V of elements and a set of subsets S of V, i.e. $\forall s \in S : s \subseteq V$, as well as integers $k, n \in \mathbb{N}$: does a subset W of S exist such that $|W| \leq k$ and $n \leq |\{v \in V \mid \exists s \in W : v \in s\}|$?

Now we consider the complexity: in short, VC can be reduced to Set Cover, and Set Cover can be reduced to Maximum Set Coverage. As VC (and Set Cover for that fact) are known to be NP-hard, Maximum Set Coverage is, too.

An instance of Set Cover corresponds exactly to an instance of Maximum Set Coverage with the exact same parameters and $n = |V|$, trivializing the reduction.

The reduction from VC on $G = (V, E)$ to SC is handled by numbering edges within E, constructing sets of incident edges for all vertices in V. These sets are all subsets of the set of numbers corresponding to all edges in E. As each such subset corresponds exactly to a node, a solution to vertex cover results in a set of subsets covering all numbers, and a solution to set cover results in a set of vertices covering all edges.

b)
An ILP that determines the CC rule may be written as follows:

$$\begin{array}{ll}
\text{maximize} & \sum_{e_j \in E} y_j \\
\text{with constraints} & \sum x_i \leq k \\
 & \sum_{S_i \in S : e_j \in S_i} x_i \geq y_j \\
 & y_j \in \{0, 1\} \\
 & x_i \in \{0, 1\}
\end{array}$$

Where $x_i = 1$ implies that subset S_i was picked, and $y_j = 1$ implies that element e_j is covered. The constraints respectively define values for x_i and y_j, limit the number of picked subsets to k, and state that for a given element e_j, the variable y_j may only be 1 if e_j is in at least one picked subset.