COMP 250 - Homework #6
Due on April 11 2017, at 23:59.

Question 1. (25 points) Greedy and Dynamic Programming Algorithms

You enter a store with N dollars in your pockets. In the store there are k different types of objects for sale: let us call then $O_0, O_2, \ldots O_{k-1}$. Each type of object is available in infinite supply but is indivisible (i.e. you can't buy a fraction of an object). Each object O_i costs C_i dollars and has utility U_i. Both are integers. The utility of object O_i is the satisfaction you get by purchasing it. Your job is to design an algorithm to plan how to spend your money in order to maximize the total utility of the objects you purchase. Your algorithm should return an array $Q=Q_0 \ldots Q_{k-1}$, where Q_i is a non-negative integer corresponding to the recommended number of objects of type i to purchase. The array Q must fit within your budget N, in other words

$$Q_0 * C_0 + Q_1 * C_1 + Q_2 * C_2 + \ldots + Q_{k-1} * C_{k-1} \leq N$$

Your goal is to maximize the total utility of your purchase, where

$$\text{Utility}(Q_0 \ldots Q_{k-1}) = Q_0 * U_0 + Q_1 * U_1 + Q_2 * U_2 + \ldots + Q_{k-1} * U_{k-1}$$

Consider for example the situation where k=4, and

$C_0 = 2$, $U_0 = 1$
$C_1 = 6$, $U_1 = 5$
$C_2 = 8$, $U_2 = 8$
$C_3 = 10$, $U_3 = 9$

Then if N=21, an optimal choice is Q_0=0, Q_1=0, Q_2=0, Q_3=2, for a total cost of $20 and a total utility of 18. Another optimal choice would be Q_0=2, Q_1=0, Q_2=2, Q_3=0.

If N=26, the optimal choice is Q_0=1, Q_1=0, Q_2=3, Q_3=0, for a total budget of $26 and a total utility of 25.

a) (8 points). Describe a greedy algorithm to try to solve the problem. Your algorithm will probably not be guaranteed to produce the optimal solution in all cases.

Algorithm GreedyChoice(int C[0…k-1], int U[0…k-1], int k, int N)
Input:

- An array C of costs for the k objects;
- An array U of utilities of the k objects;
- A total budget N.

Output: An array Q[0…k-1] of quantities to purchase.

A reasonable greedy algorithm would be to pick first the object with the largest ratio of utility to cost.

```
nbPicked=0
while (N>0)
        bestObject = null
        bestRatio = 0
        for i = 0 to k-1
                if (C[i]<=N and U[i]/C[i] > bestRatio)
                        bestRatio = U[i]/C[i]
                        bestObject = i
        Q[bestObject] = Q[bestObject] + 1
        N = N – C[i]
```

b) (4 points) Give an example of a situation (choice of C, U, and N) where your greedy algorithm fails to produce the optimal solution.

Suppose there are two types of objects, with
$U_0 = 10$, $C_0 = 9$. Ratio = 1.11
$U_0 = 8$, $C_0 = 8$. Ratio = 1.0

Suppose the total budget is N = 17
The greedy algorithm would pick one object of type 0 and nothing else, for a total utility of 10. A better solution would be to pick two objects of type 1, for a total utility of 16.

c) (10 points) Write a dynamic programming algorithm to solve the problem. Your algorithm has to work for any choice of C, U, k, and N. First write the recursive formula for Utility(*N*), the optimal utility that can be achieved within a maximum cost *N*. and then give the pseudocode for the dynamic programming algorithm.

Algorithm DynProgChoice(int C[0…k-1], int U[0…k-1], int k, int N)
Input:

- An array C of costs for the k objects;
- An array U of utilities of the k objects;
- A total budget N.

Output: The maximum total utility that can be achieved with a budget N
For a 5 point bonus: An array Q[0…k-1] of quantities to purchase.

Recursive formula:

$$Utility(N) = \begin{cases} \max\{ Utility(N-C_0) + U_o, Utility(N-C_1) + U_1, \ldots, Utility(N-C_{k-1}) + U_{k-1} \} & \text{if } N>0 \\ 0 & \text{if } N<=0 \end{cases}$$

Algorithm:

```
UtilityArray = new int[N+1]
for i = 0 to N
        maxvalue=0
        maxindex=0
        for j = 0 to k
                if (C[j]<= i and U[j] + UtilityArray[i-j] > maxvalue)
                        maxvalue =  Utility(N-C_1) + U_1
                        maxindex = j
        UtilityArray[maxindex] = UtilityArray[maxindex] + 1

Return UtilityArray[N]
```

In order to actually recover the choice of objects that achieves the maximum utility, we basically trace back the computation of the Utility

```
while (N>0)
        for i = 0 to k-1
                if (C[i]<= N and UtilityArray[N] = UtilityArray[N – C[i]] + U[i] {
                        Q[i] = Q[i+1]
                        N = N – C[i]
                        Break
```

d) (3 points) Use your algorithm to calculate the optimal purchase for N=38, using the cost and utility values given in the example.

The optimal solution is to choose Q[0]=0, Q[1]=1, Q[2]=4, Q[3]= 0, for a total cost of 38 and a total utility of 37.

Question 2. (15 points) Sorting, one last time!

a) (10 points) Consider the following sorting problem: You are given an array of n integers. The integers are all between 1 and 2n, but are not necessarily all distinct. Write an algorithm that sorts this array and that runs in time O(n) in the worst case. (Hint: this is really easy once you see it. I could have said that the numbers are all between 1 and 10 n, or between 1 and 9394923 n and it would make no difference, except that the constant hidden in the big-Oh notation would get larger.)

Solution: This algorithm is called counting-sort.

Algorithm counting-sort(A, n)
Input: an array A of n storing distinct integers between 1 and 2n
Output: The array A is sorted

Let X[0...2n] be an array of booleans.
Initialize all elements of X to false

```
for i = 0 to n-1 do
    X[ A[i] ] = true
k = 0
for i = 1 to 2n do
    if ( X[i] ) then
        A[k] = i
        k = k +1
```

b) (5 points) Why can't your algorithm be used efficiently to sort an arbitrary set of integers?

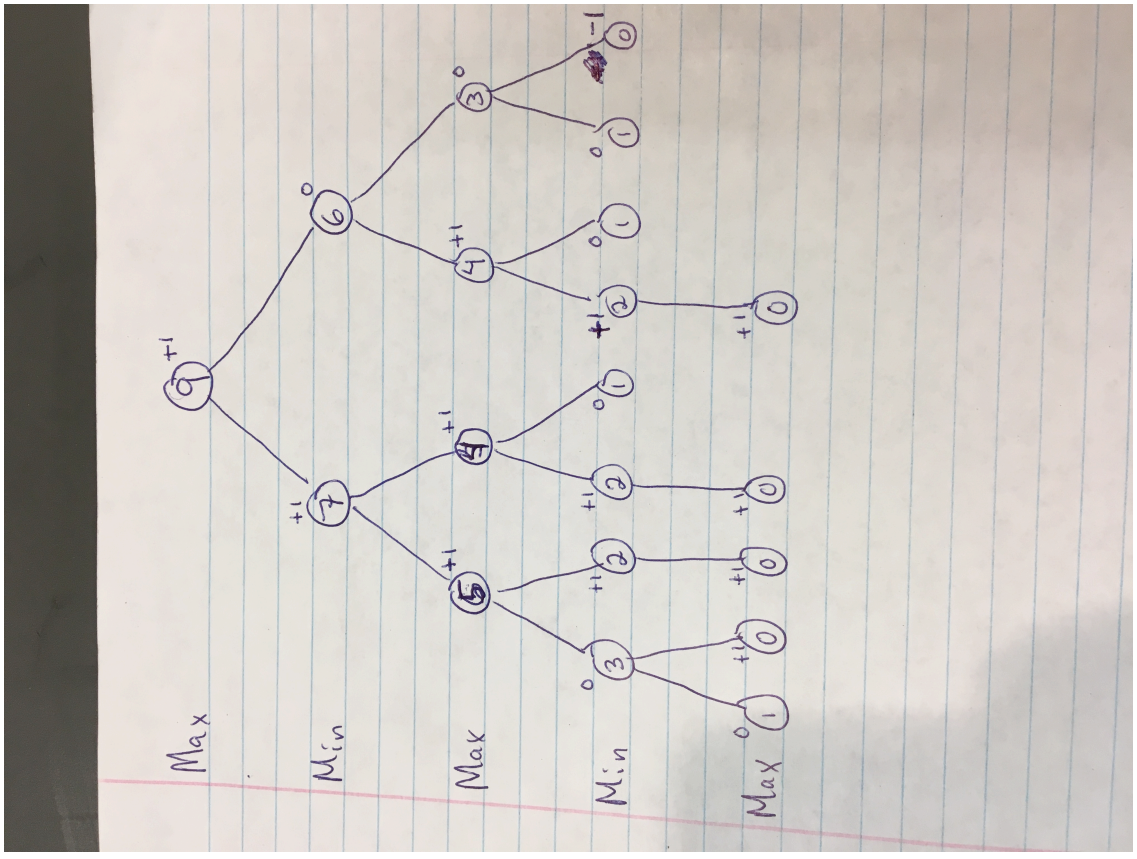
Solution: We need an array as large as the largest element that is in the input array A. If the largest element of A is very large, the array X needed will be very large too and the running time of the second loop will be poor. For example, if the numbers are between and 1 and n^2, the running time would be $O(n^2)$, because the array X would be of size n^2.

Question 3. (20 points) Game trees

Consider the following two-player game that is a variant of the popular Nim game. The game starts with a stack of n matches. Players alternate in removing matches from the stack. Each player can remove from the stack either 2 or 3 matches (provided there are sufficiently many matches left). The player who removes the last match loses, except if there is only one match left, in which case the game is a draw. For example, if n=4, then the first player wins by removing two matches, leaving the opponent with no choice but to choose the two remaining matches. If n=5, then the first player wins by picking 3 matches. If n=6, then the game results in a draw.

a) **(10 points)** Draw the game tree for a game starting with n=9 matches. If both players play as well as possible, who will win?

Answer: Player #1 wins



b) **(10 points)** In general, if the game starts with n matches, who will win the game? Express your answer as a function of n.

Let us consider small values of n, to see if there is a pattern:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	0	-1	0	1	1	0	-1	0	1	1	0	-1	0	1	1	0

We guess the pattern:
If n%5 = 1 or 3, then it is a draw
If n%5 = 0 or 4, then player 1 wins
If n%5 = 2, then player 2 wins.

Question 4. (40 points) Graph algorithms

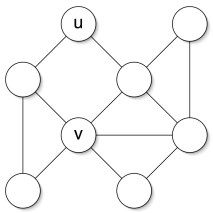
For this question, assume a graph has *n* vertices numbered 0,1, 2,...*n*-1.
In your algorithm, you can use the following graph ADT methods:
• *getNeighbors*(vertex v) returns the set of neighbors of vertex *v*. It is fine for you to write something like: for each vertex v in *getNeighbors*(u) do ...
• boolean *getVisited*(vertex v) returns TRUE if and only if vertex *v* has been marked as visited.
• *setVisited*(vertex *v*, boolean *b*) sets the visited status of vertex *v* to *b.*
• *getColor*(vertex *v*) returns the color of vertex *v*, either 0 or 1. If the color of vertex *v* has not been set previously, *getColor*(*v*) is undefined.
• *setColor*(vertex *v*, color *c*) sets the color of vertex *v* to *c*.

You may also want to associate to each vertex an integer called distance, which can be set and accessed through

- *setDistance*(vertex *v*, int *d*) sets the distance stored in *v* to *d*
- *getDistance*(vertex *v*) returns the distance stored in *v.* Undefined if that distance has not been set previously.

a) (20 points) In an undirected connected graph *G=(V,E),* the distance *d(a,b)* between vertices *a* and *b* is the number of edges in the shortest path between *a* and *b*. The excentricity of a vertex *a* is defined as the largest shortest-path distance between vertex *a* and any other vertex:
excentricity(*a*) = max { *d(a,b)* : $b \in V$ }

For example, in the graph to the right, excentricity(*u*) = 3 and excentricity(*v*) = 2.



Problem: Write an algorithm to compute the excentricity of a given vertex in a graph.

Algorithm excentricity(vertex *u*)
Input: a vertex *u* from the graph
Output: the excentricity of *u*

```
// We use a modified breadth-first search
    q ← new Queue()
    setVisited(u, true)
    setDistance(u,0)
    q.enqueue(u)
    maxDistance = 0
    while (! q.empty()) do
        w ← q.deque()
        maxDistance = getDistance(w)
        for all  v ∈ w.getNeighbors() do
                if  ( ! getVisited(v) ) then
                        setVisited(v, true)
                        setDistance(v, 1 + getDistance(w))
                        s.enqueue(v)
   return maxDistance
```

b) (20 points) Determining if an undirected graph can be colored with only three colors is an NP-complete problem. However, determining if it can be colored with only *two* colors is much easier. Write the pseudocode of an algorithm that determines if the vertices of a given connected undirected graph can be colored with only two colors (named 0 and 1) so that no two adjacent vertices have the same color.

Algorithm is2colorable(vertex u)
Input: a graph vertex u
Output: true if the graph to which u belongs is 2-colorable, and false otherwise
/* WRITE YOUR PSEUDOCODE HERE */

Solution: We use a modifier depth-first search, were we alternate colors when we follow edges. If we get to a vertex that is already visited and of the same color as the previous vertex, then the graph is not 2-colorable.

```
Algorithm checkTwoColorable(Graph G, vertex v, color c)
Input: A graph G and a vertex v to be colored with color c (either 0 or 1)
Output: True if the graph is 2-colorable, False otherwise

setVisited(v, TRUE)
setColor(v, c)
for each w in getNeighbors(v) do
        if (w.getVisited() = TRUE) then
                if (w.getColor() = v.getColor() ) return FALSE
        else
                if (checkTwoColorable(G, w, 1-c)=FALSE) return FALSE
```

return TRUE