COMP 250 Assignment #6 – Written Responses

Alex Hale

1) a)

```
Algorithm int[] GreedyChoice(int C[k], int U[k], 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[k] of quantities to purchase
int[] Q = new int[k];
while (N > min(C)) {                    // iterate while we can still afford the cheapest item
    int highestUtility = 0;
    int highestUtilityIndex = -1;
    for (int i = 0; i < k; i++) {       // find the object with the highest utility
        if (C[i] > N) {                 // if the cost of an item is too high for our remaining budget,
            U[i] = 0;                       // remove it from consideration
        } else if (U[i] > highestUtility) {
            highestUtility = U[i];
            highestUtilityIndex = i;
        }
    }
    // buy as many of the item with the max utility as we can afford
    int numberPurchasable = N / C[highestUtilityIndex];
    Q[highestUtilityIndex] += numberPurchasable;
    N -= numberPurchasable * C[highestUtilityIndex];
}
return Q;
```

1) b)

N = 10

C = {5, 2}

U = {3, 2}

- greedy algorithm picks 2 of the first item => Utility = 6
- optimal solution is 5 of the 2nd item => Utility = 10
    - greedy algorithm picked non-optimal solution

1) c)

Recursive formula: Utility(N) = max((U(0) + Utility(N - C(0)) AND N >= C(0) + cost(N - C(0)),
(U(1) + Utility(N-C(1)) AND N >= C(1) + cost(N - C(1)),
...
(U(k) + Utility(N-C(k)) AND N >= C(k) + cost(N - C(k)))

- U(1...k) and C(1...k) are "base cases"
- the utility at budget N is equal to the utility of one of the base cases plus the utility at N less the cost of that base case, so long as the cost of the "new" base case plus the cost of the "subsidiary" case is less than or equal to the budget N

Algorithm DynProgChoice(int C[k], int U[k], 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[k] of quantities to purchase
// I did the bonus here - if the output were the maximum total utility with budget N, I would return utility[N]

```
int[] utility = new int[N+1];                  // maximum total utility possible at budgets 0, 1, 2, ..., N
int[] cost = new int[N+1];                     // cost used to achieve max total utility (cost[i] <= i)
boolean[][] additionalQ = new boolean[N][k];  // which additional object is purchased at each value of N
utility[0] = 0;
cost[0] = 0;
for (int i = 0; i < N; i++) {
        for (int j = 0; j < k; j++) {
                additionalQ[i][j] = false;           // preset everything to false
        }
}
for (int i = 1; i <= N; i++) {                   // iterate from 0 up to N   (i = the budget at this step)
        int maxUtil = 0;            // maximum added utility
        int maxUtilIndex = -1;              // which base case gives this maximum added utility
        for (int j = 0; j < k; j++) {           // iterate through all k base cases
                int thisUtil = 0;
                if (i >= C[j]) cost[i] = C[j] + cost[i - C[j]];            // use this spot in the array as a temp variable
                else cost[i] = 0;

                if (i >= cost[i]) {                        // if our budget allows us to buy one more of this object
                        thisUtil = U[j] + utility[i - C[j]];             // get the utility for this budget + this base case
                        if (thisUtil > maxUtil) {
                                maxUtil = thisUtil;                  // set this utility to the maxUtil if it's higher
                                maxUtilIndex = j;                    // than the current maxUtil
                        }
                }
        }
        utility[i] = maxUtil;
        if (maxUtilIndex > -1) {
                // cost to purchase new object AND objects from previous case
                cost[i] = C[maxUtilIndex] + cost[i - C[maxUtilIndex]];
                // save which base case was added (for future use)
                additionalQ[i][maxUtilIndex] = true;
        } else {cost[i] = 0;}
}
int[] Q = new int[k];
while (cost[N] > 0) {                // iterate while we can still afford the cheapest item
        for (int i = 0; i < k; i++) {
                if (additionalQ[cost[N]][i]) {                // find how many times each item was purchased
                        Q[i]++;
                        cost[N] -= C[i];                     // move down to next item
                }
        }
}
return Q;
```

1) d)

Q = {0, 1, 4, 0} => U(38) = 4*8 + 1*5 = 37

- therefore, maximum utility for N = 38 is 37

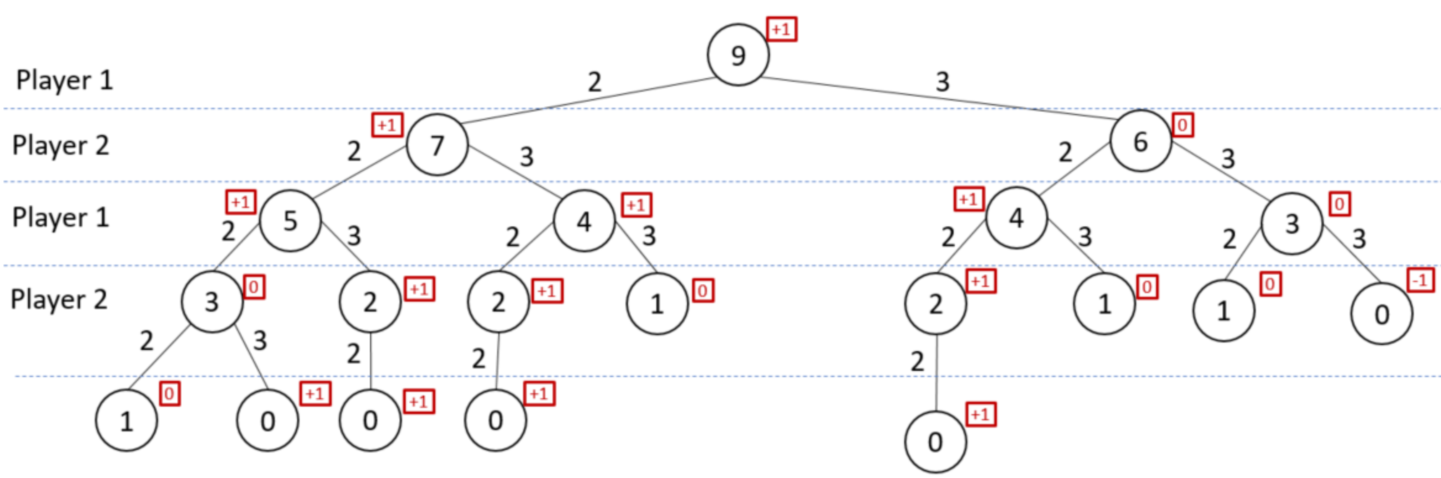
2) a)

```
Algorithm sort(int[] array)
Input: an array of n integers, where each integer is between 1 and 2*n
Output: a sorted array of n integers
int n = array.length;
int[] numberOfEachNumber = new int[2*n];           // make new array of length 2n
for (i = 0 to n-1) do {
        // from the input array, count the number of 1s, the number of 2s, ... , the number of 2ns
        numberOfEachNumber[array[i] - 1]++;
}                                                  // runs in time O(2n) => O(n)
int i = 0;
int j = 1;
while (j <= 2*n) {                           // iterate until the end of the numberOfEachNumber array
        while (numberOfEachNumber[j-1] > 0) {
                // iterate until we've placed all the 1s, then all the 2s, ... , then all the 2ns
                array[i] = j;                                      // runs in time O(2n) => O(n)
                numberOfEachNumber[j-1]--;
                i++;
        }
        j++;
}
return array;                                      // Total time is O(n) + O(n) + constants => O(n)
```

2) b)

- Without having a cap on the size of the largest integer in the array, we can't define the size of numberOfEachNumber, and so we can't use this approach efficiently.
- To use this approach, we would first have to run through the array once to find the largest integer: inefficient!

3) a) Player 1 will win if the game starts at 9 matches and both players play optimally.



- in the diagram above and the answer below:
  - +1 => winning position for Player 1
  - -1 = > winning position for Player 2
  - 0 => position where a draw will occur
    - Assuming both players will play optimally

n	Position
0	no game
1	0
2	-1
3	0
4	1
5	1
6	0
7	-1
8	0
9	1
10	1
11	0
12	-1
13	0
14	1
15	1
16	0
17	-1

3) b)

$$Winner(n) = \max\left(\left(-Winner(n-2)\right), \left(-Winner(n-3)\right)\right)$$

- This is a recursive formula
  - $Winner(n) = +1$ => player who goes first wins
  - $Winner(n) = -1$ => player who goes second wins
  - $Winner(n) = 0$ => game is a draw

- If a non-recursive formula is needed:
  - There is a repeating pattern (as seen in the chart to the right), so a piecewise function could be made using the variable n

$Winner(n) = +1$ when $n = 4, 5, 9, 10, 14, 15, 19, 20, \dots$
$Winner(n) = -1$ when $n = 2, 7, 12, 17, \dots = 2 + 5k$, where k is a positive integer
$Winner(n) = 0$ when $n = 1, 3, 6, 8, 11, 13, 16, 18, \dots$

4) a)

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

```
s <- new Stack();                    // stack to store our path
setVisited(u, true);           // set values of starting node
setDistance(u, 0);
s.push(u);
int eccentricity = 0;          // variable to track highest eccentricity
while (!s.empty()) do {
        w <- s.pop();
        parentDepth <- w.getDistance();
        for each vertex v in getNeighbours(w) do {
        // iterate through all the neighbours of the node in question
                if (!getVisited(v)) {
                        // if a node hasn't yet been visited:
                                // set it to visited, and give it a distance one greater than its parent
                        setVisited(v, true);
                        setDistance(v, parentDepth+1);
                        // if this node is the deepest we've seen so far, set it to the new eccentricity
                        if (getDistance(v) > eccentricity) eccentricity = getDistance(v);
                        // add this node to the stack so we can check its neighbours
                        s.push(v);
                }
        }
}
return eccentricity;    // return the highest distance found
```

4) b)

```
Algorithm is2colorable(vertex u)
Input: a vertex u from the graph
Output: true if the graph to which u belongs is 2-colorable, false otherwise
frontier <- new Queue()             // Queue to store our path
setColour(u, 0)                     // set values of starting node
setVisited(u, true)
frontier.enqueue(u)
while (!frontier.empty()) do {
    s = frontier.dequeue();
    parentColour = getColour(s);
    for each vertex v in getNeighbours(s) do {
    // iterate through all the neighbours of the node in question
        if (!getVisited(v)) {
            if (parentColour == 0) setColour(v, 1)
            else setColour(v, 0);
            // if the neighbour hasn't yet been visited:
                // set it to the opposite of the parent's colour, set it visited, add it to the queue
            setVisited(v, true);
            frontier.enqueue(v);
        } else {
            // if a neighbouring node has already been visited AND that neighbouring node
            // has the same colour, return false
            if (getColour(v) == getColour(s)) return false;
        }
    }
}
return true;        // we didn't encounter any neighbouring nodes with the same colour, so return true
```