# COMP 250 - Homework #6
# Due on Dec. 3rd 2014, at 23:59.

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

You are leaving on a hiking expedition and you want to fill your backpack with useful objects. You have n different types of objects to choose from. Each object of type i has a integer weight W(i) and an integer value V(i). Your backpack can carry at most a total weight of T. Your goal is to find the optimal total value Opt(T) you can put in your backpack subject to this constraint Assume that you have infinite supply of each type of object, so you can put as many copies of each object as you want. However, the items are not divisible; you have to take an integral number of each (either zero, one, two, etc.).

For example, for n = 4, we might have
W(0) = 20, V(0) = 9
W(1) = 15, V(1) = 7
W(2) = 11, V(2) = 5
W(3) = 8, V(3) = 1

Then if T = 34, the optimal solution is to take three item2 and zero of all others, for a total weight of 33 ≤ T and a total value of Opt(T) = 15. If T = 35, the optimal solution is to choose one item0 and one item1, for a total weight of 35 ≤ T and a total value of Opt(T) = 16.

a) (4 points) What is the value of Opt(41)?

*Opt(41) = 19, which is obtained by choosing two objects of type 1 and one object of type 2, for a total weight of 2*W(1) + W(2) = 41 and a total value of 2*V(1)+V(2) = 19.*

b) (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.

*Perhaps the most reasonable greedy algorithm would be one where we repeatedly select the item with the largest ratio of value to weight, provided selecting such an object doesn't bust our weight limit. In the above example, with T=41, this would yield:*
*Item1 (ratio = 7/15 = 0.466, weight=15)*
*Item1 (ratio = 7/15 = 0.466, weight = 15)*
*Item2 (ratio = 5/11 = 454, weight = 11)*
*Which the optimal solution.*

c) (4 points) Give an example of a situation where your greedy algorithm fails to produce the optimal solution.

*For T=20, we would get:*
*Item1 (ratio = 7/15 = 0.466, weight=15)*
*and then we would be unable to add anything to our backpack. This solution is clearly inferior to choosing a single item 0.*

d) (9 points) Write a dynamic programming algorithm to solve the problem. Your algorithm has to work for any choice of W and V. First write the recursive formula, and then give the pseudocode for the dynamic programming algorithm.

***Solution**: Let X(w) be the maximal value of a backpack of weight at most w.*
*Then, we have the following equation for X(w):*

$$X(w) = \begin{cases} 0 & \text{if } w=0 \\ \max \{ V(i) + X(w-W(i)) : i=1...n \text{ and } w>= W(i)\}, \text{ or 0 if this set is empty} & \text{if } w>0 \end{cases}$$

*Thus, the dynamic programming algorithm solution is:*

***Algorithm** BackpackValue(W, V, n, T)*
***Input**: An array W of weights and an array V of values, both of them of size n.*
***Output:** Returns the optimal value of the backpack. Doesn't need to output what choice of items actually realizes that value.*

```
Let X be an array of size T
Let choice be an array of size T used to remembre the first object to choose for each
                                                                    possible weight
for w = 0 to T do
   // compute this maximum
  X[w] = 0     // the maximum will always be at least zero
  choice[w] = 0   // will be used to save
  for i = 1 to n do
     if (w>= W[i] AND V[i] + X[ w-W[i] ] > X[w]) then
        X[w] = V[i] + X[ w-W[i] ]
        choice[w] = i

// the score of the best solution is X[T]
// print the choice of items leading to that choice
w = T
```

```
while (choice[w] > 0) do
    print "Choose item " + choice[w]
    w = w – W[ choice[w] ]
return X[T];
```

# 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?

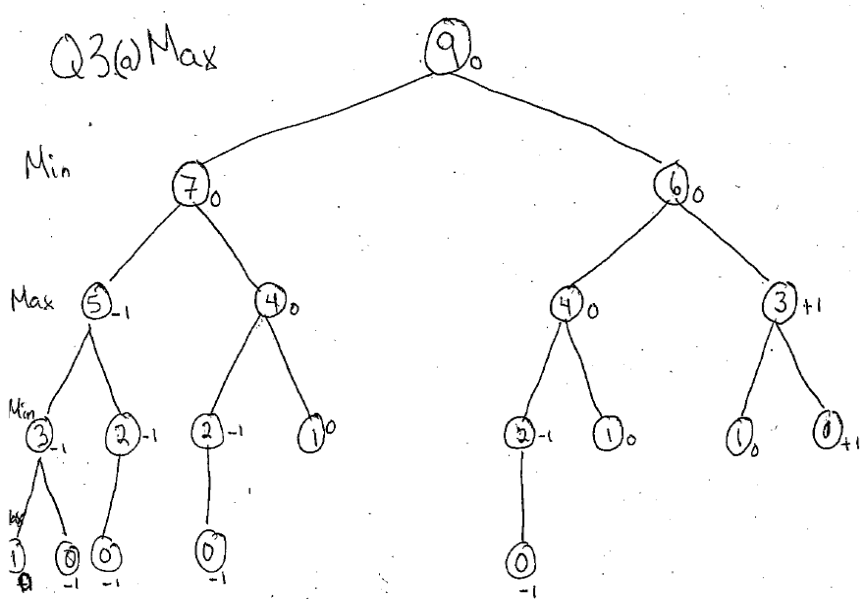
*Solution: We need an array as large as the largest element that is in the input array A. If the largest element of A is very large, the array X needed will be very large too and the running time of the second loop will be poor. For example, if the numbers are between and 1 and $n^2$, the running time 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 alternately remove 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 wins, except if there is only one match left, in which case the game is a draw. For example, if n=3, then the first player wins by removing all three matches. If n=5, then the first player necessarily loses because any move he makes leads to a situation where he loses.

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: The game is a tie.*




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* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *0* | *1* | *1* | *0* | *-1* | *0* | *1* | *1* | *0* | *-1* | *0* | *1* | *1* | *0* | *-1* |

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

## **Question 4.** (15 points) Graphs algorithms

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. Assume the graph has *n* vertices numbered 0,1, 2,...*n*-1. Use the following graph ADT methods:
• *getNeighbors*(int *i*) returns the set of neighbors of vertex *i*. It is fine for you to write something like: for each vertex v in *getNeighbors*(17) do ...
• boolean *getVisited*(int *i*) returns TRUE if and only if vertex *i* has been marked as visited.
• *setVisited*(int *i*, boolean *b*) sets the visited status of vertex *i* to *b.*
• *getColor*(int *i*) returns the color of vertex *i*, either 0 or 1. If the color of vertex *i* has not been set previously, *getColor*(*i*) is undefined.
• *setColor*(int *i*, color *c*) sets the color of vertex *i* to *c*.

*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
```