Homework 6

1.

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.

Choose as many items that has largest value to weight ratio as possible, than as many items that has second largest value to weight ratio as possible, than third….

Value to weight ratio for:

Item0=V(0)/W(0)=9/20=0.45

Item1=V(1)/W(1)=7/15=0.467

Item2=V(2)/W(2)=5/11=0.454

Item3=V(3)/W(3)=1/8=0.125

So item1 is more favorable than item2, which is more favorable than item0, which is more favorable than item3.

Let N(0), N(1), N(2), N(3) be the number of item0, item1, item2, item3 to put in backpack, T be the maximum weight the backpack can carry.

N(1)=(int)(T/W(1))

N(2)=(int)((T-W(1)\*N(1))/W(2))

N(0)=(int)((T-W(1)\*N(1)-W(2)\*N(2))/W(0))

N(3)=(int)((T-W(1)\*N(1)-W(2)\*N(2)-W(0)\*N(0))/W(3))

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

When T=20, the greedy algorithm gives a solution of 1 item1 (total value=7), but the optimal solution is 1 item0 (total value=9).

c) (10 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.

Recursive formula:

Opt(T)=0 for T=0

Opt(T)=max{Opt(T-W(item j))+V(item j) | j=0,1,2…n, W(n)<=T}

Pseudocode:

Algorithm chooseItems(W[0…n-1],V[0..n-1],T)

Input: an array W containing the weights of the items, an array V containing the values of the items, and integer T which is the maximum weight the backpack can carry.

Output: the optimal total value you can put in your backpack.

int Opt[]=new int[T+1]; //Opt[0…T]

Opt[0]=0;

for i =1 to T do

maxValue=-1;

for j=0 to n-1 do

if (W[j]<=i) then maxValue=max(maxValue, Opt[i-W[j]]+V[j]);

Opt[i]=maxValue;

return Opt[T];

d) (3 points) Use your algorithm to calculate Opt(37).

Opt(37)=17

2.

**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. )

Algorithm sort(A[n])

Input: an array of n integers to be sorted, the integers are all between 1 and 2n.

Output: an array of sorted n integers.

int B[]=new int[2n+1];

for i=1 to 2n do

B[i]=0;

for i=0 to n-1 do

B[A[i]]++;

m=0;

for i=1 to 2n do

while (B[i]>0) do

A[m]=i;

B[i]--;

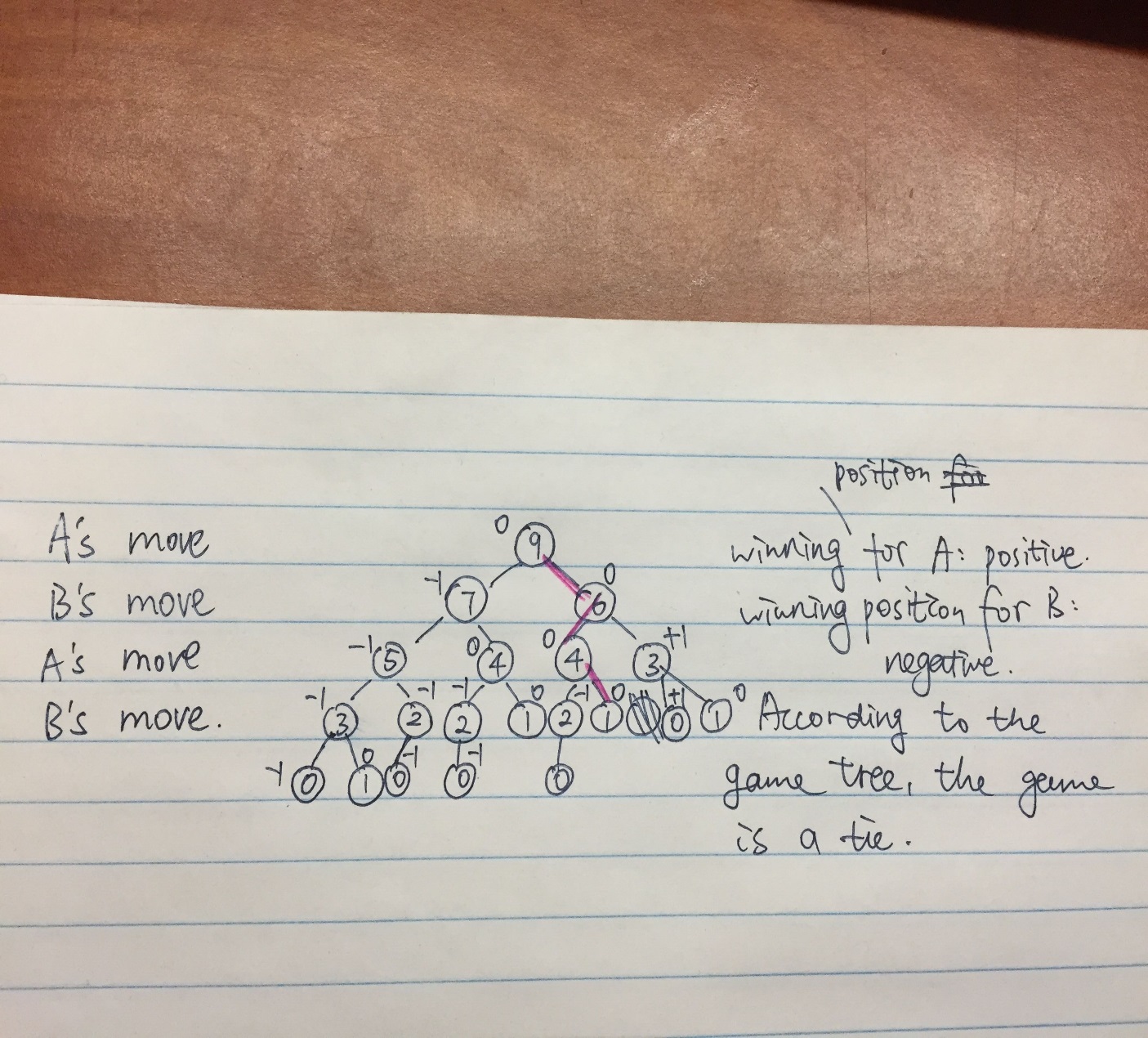
return A;

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

In order to sort a set of integers, this algorithm require an array as large as the largest element in the input array. For arbitrary set of integers, we are not sure about the largest element, so we need an array that contains all possible integers, and it’s very time-consuming to loop through this big array.

3.

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?



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.

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

According the the chart, we can infer that

if n%5=0, player B wins.

If n%5=2 or 3, player A wins.

If n%5=1 or 4, it is a draw.

4.

a)

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

//I divided the question into two algorithms.

Algorithm distance(vertex u)

Input: a vertex u from the graph.

Precondition: every vertex in the graph is initialized to 0 distance and unvisited statue.

Postcondition: every vertex in the graph is labeled with its distance from the vertex u.

setVisited(u,true);

array A←getNeighbors(u);

for a in A do

if (!getVisited(a) || getDistance(u)+1<getDistance(a)) do

setDistance(a,getDistance(u)+1);

distance(a);

**Algorithm** excentricity(vertex *u*)

**Input:** a vertex *u* from the graph.

Precondition: every vertex in the graph is initialized to 0 distance and unvisited statue.

**Output:** the excentricity of *u*

distance(u);

q←new Queue();

setVisited(u,false); //since all we call the method distance(u), all the vertices have been set to visited, so this time we set to unvisited

q.enqueue(u);

m=0;

while(!q.empty())do

w←q.deque()

m=max(getDistance(w),m);

for all a∈getNeighbors(w)

if getVisited(a)==true do

setVisited(a,false);

q.enqueue(a);

return m;

b)

**Algorithm** is2colorable(vertex u)

**Input:** a graph vertex u

**Output:** true if the graph to which u belongs is 2-colorable, and false otherwise

q←new Queue();

setVisited(u,true);

setColor(u,0);

q.enqueue(u);

while(!q.empty())do

w←q.deque()

for all a∈getNeighbors(w)

if (!getVisited(a)) then

setVisited(a,true);

setColor(a,1-getColor(w))

q.enqueue(a);

else

if(getColor(a)==getColor(w) return false;

return true;