CSCD320 Homework6, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 25, 2014 (Sunday)

Please follow these rules strictly:

1. Write your name and EWUID on **EVERY** page of your submission.

2. Verbal discussions with classmates are encouraged, but each student must independently write his/her own solutions, without referring to anybody else's solution.

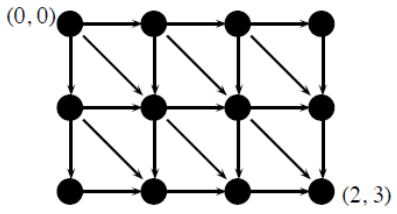
3. The deadline is sharp. Late submissions will **NOT** be accepted (it is set on the Blackboard system). Send in whatever you have by the deadline.

4. Submission must be computer typeset in the **PDF** format and sent to the Blackboard system. I encourage you all to use the LaTeX system for the typesetting, as what I am doing for this homework as well as the class slides. LaTeX is a free software used by publishers for professional typesetting and are also used by nearly all the computer science and math professionals for paper writing.

5. Your submission PDF file must be named as:
firstname_lastname_EWUID_cscd320_hw6.pdf
(1) We use the underline ' ' not the dash '-'.
(2) All letters are in the lower case including your name and the filename's extend.
(3) If you have middle name(s), you don't have to put them into the submission's filename.

6. Sharing any content of this homework and its keys in any way with anyone who is not in this class of this quarter is NOT permitted.

Problem 1 (25 points)**.** *Consider a directed graph arranged into rows and columns. Each vertex is labeled* (j, k)*, where row* j *lies between* 0 *and* M*, and column* k *lies between* 0 *and* N*. The start vertex is* (0, 0) *and the destination vertex is* (M,N)*. Each vertex* (j, k) *has an associated weight* W(j, k)*. There are edges from each vertex* (j, k) *to at most three other vertices:* (j + 1, k)*,* (j, k + 1)*, and* (j + 1, k + 1)*, provided that these other vertices exist*



An example graph with $M = 2$ *and* $N = 3$.

The input of your algorithm is the value of non-negative integers M *and* N*, as well as a 2-d array* W[0 . . .M × 0 . . .N]*, where* W[j, k] *stores the weight of the node at coordinates* (j, k)*.*
Your goal is to count the number of different paths from (0, 0) to (M, N) that have minimum total weight. The weight of a path is defined as the total weights of all the nodes on that path.

CSCD320 Homework6, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 25, 2014 (Sunday)

1. Provide a recursive formula that computes the exact value that is specified in the problem. Check the book chapter on dynamic programming for an example of such recursive formular, which essentially catches the understanding of the recursive structure in the solution.

2. Express the recursive formula as a bottom-up and iteration-based dynamic programming algorithm, and determine its running time in big-oh notation. Make your bound as tight as possible.

1. Recursive formula:

```
Global Count;
findNumberOfMinWeightPaths(M, N, W){
        q = ∞
        min = ∞
        If(N == 0 && M== 0){
                return W[M][N]
        }

        if(N == 0 && M != 0){
                q = min(q, W[M][N] + findNumberOfMinWeightPaths(M-1, N, W))
        }
        else if(N != 0 && M == 0){
                q = min(q, W[M][N] + findNumberOfMinWeightPaths(M, N-1, W))
        }
        else{
                q = min(q, W[M][N] + findNumberOfMinWeightPaths(M-1, N, W))
                if(q < min)
                        count = 1
                        min = q
                else if( q == min)
                        count++
                q = min(q, W[M][N] + findNumberOfMinWeightPaths(M-1, N-1, W))
                if(q < min)
                        count = 1
                        min = q
                else if( q == min)
                        count++
                q = min(q, W[M][N] + findNumberOfMinWeightPaths(M, N-1, W))
                if(q < min)
                        count = 1
                        min = q
                else if( q == min)
                        count++
        }
}
```

This algorithm starts at the last element in the input 2d array, and recursively calls back to each possible path from the current element. It adds the path weights as the calls return, and keeps track of the overall minimum path found so far. There is a global count that will be set to one if a new minimum is found or increment by one if a path is found that is equivalent to the current minimum.

CSCD320 Homework6, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 25, 2014 (Sunday)

2. Iteration Based, Bottom Up Formula:

```
findNumberOfMinWeightPaths(M, N, W){

        min = ∞
        count = 0
        r[M][N] = new int array of zeros

        for(j = 0; j < M ; j++){

                for(k=0; k < N; k++){
                        if(j == 0 && k == 0)
                                r[j][k] = W[j][k]
                        else if(j == 0)
                                r[j][k] = W[j][k] + r[j][k-1]
                        else if(k == 0)
                                r[j][k] = W[j][k]+r[j-1][k]
                        else
                                r[j][k] = min(W[j][k]+r[j-1][k],W[j][k]+r[j-1][k-1],W[j][k]+r[j][k-1])

                        if(r[j][k] < min){
                                min = r[j][k]
                                count = 1
                        }
                        else if(r[j][k] == min)
                                count++
                }

        }
        return count
}
```

This algorithm walks through the input 2d array start to finish, and stores the lightest weight path to each element in another array as it travels. It has a count for the number of minimum weight paths. It will keep track of the overall min path found so far as it travels, and if it find a smaller path it will reset the count at one. It if finds a path weight equal to the current min path, then it will increment the count up by one. The algorithm only touches each element once. If the amount of elements "n" in the input array is the problem size, then this takes O(n) time.

CSCD320 Homework6, Spring 2014 Eastern Washington University. Cheney, Washington.

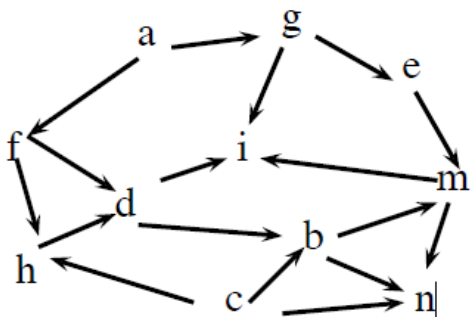
Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 25, 2014 (Sunday)

Problem 2 (15 points)**.** *We have learned that the greedy strategy does not work well for the 0-1 knapsack problem. Now you are asked to self-study the backtracking strategy based solution for the 0-1 knapsack problem. After you have had a good understanding of the solution, you need to precisely and concisely restate the idea of the solution in YOUR own language, including but not limited to how the search procedure is organized and what is the clever idea for pruning the search space. Show the source of your reading, such as the url of the webpages, the title and page of a book, the title/author/year of an article, etc. Note: If you just copy and paste with minor changes in wording without your own understanding, you will get zero for this problem.*

To solve the 0-1 knapsack problem, each item will be sorted into a state space tree in order based off of highest price/weight. For each node, the right child node will be a duplicate of its parent in terms of weight and price values. Each left child node will contain the weight and price subtotals of the next biggest price/weight item, added to the subtotals of the weight and price of all the nodes before it that are also left child nodes. After the tree is sorted in this manner, we can perform a depth first search to calculate each possible combination of items that will fit in the knapsack, and take the combination that has the highest total value. We can prune the tree by keeping track of max profit we have found so far that fits into the knapsack; and as we search if we step to a child node whose weight will fill the knapsack and whose value is not as big as the max profit found so far, we can simply backtrack from there and skip all of the nodes underneath.

Source: http://wireless.cs.tku.edu.tw/~kpshih/course/Algorithms/Chapter%205.pdf

Problem 3 (10 points)**.** *Print the BFS and DFS (the sequences of letters being visited over the course of the search) that starts from the vertex* d *of the following graph. If a node has multiple next-hops, then search the next-hops in the order of their vertical coordinates from the lower ones to the higher ones. For example, node* `b` *has a lower vertical coordinate than the node* `i`*. (Note that the textbook's DFS algorithm tries all vertices as the starting node, but here you only need to show the print of the DFS that starts from the vertex* d*).*



BFS: d, b, i, n, m

DFS: d, b, n, m, i

CSCD320 Homework6, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 25, 2014 (Sunday)

Problem 4 (15 points)**.** *Given the adjacency list representation of an unweighted graph* G = (V,E)*, give your pseudocode that constructs the matrix representation of* G*. Describe the time complexity of your algorithm in the big-oh notation.*

```
convertToMatrix(adjList)
{
        int V = adjList.length
        int matrix = new int [V] [V]
        for(i = 0; i < V; i++){
                for(j = 0; j < V; j++){
                        matrix[i][j] = 0
                }
        }

        Node check = null

        for(i = 0; i < V; i++){
                for(j = 0; j < V; j++){
                        check = adjList[i]
                        if(adjList[i] == adjList[j]){
                                j++
                        }
                        while(check != null && check != adjList[j]){
                                check = check.next
                        }
                        if(check != null){
                                matrix[i][j] = 1
                        }
                }
        }
}
```

The first double for loop of this code, used to initialize every value of the matrix to zero, will take O(V*V) time. Within the second double for loop(V*V), a linked list will be traveled and in worst case each linked list will have V links, so this will take O(V*V*V) time. So altogether this will be $O(V^2+V^3) = O(V^3)$.

CSCD320 Homework6, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 25, 2014 (Sunday)

Problem 5 (15 points)**.** *The DFS algorithm that we discussed in class uses the adjacency list representation of a graph* G = (V,E) *and its time cost is* O(|V |+|E|)*. Suppose you are only given the matrix representation of* G*, describe your pseudocode for DFS of* G *using the matrix representation. Give the time complexity of your algorithm in the big-oh notation. Did you learn why we used the adjacency list representation for DFS in class ? Note: you can assume* matrix[i, j] = 1 *if* (i, j) ϵ E*;* matrix[i, j] = 0*, otherwise.*

Global int V = number of verticies
Global bool seen[V] = false for all values

```
dfs(G,i){
	seen[i] = true
	for(int j = 0; j < V; j++){
		if(!seen[j] && G[i][j] == 1){
			print G[i][j]
			dfs(G, j)
		}
	}
}
```

In the worst case, this algorithm will have to iterate through the entire matrix, and do allot of wasteful checking to see if (i,j) is a one, and if j has already been seen before. This mean that it will take $O(V^2)$ time, instead of the significantly faster O(|V|+|E|) time needed for the DFS of an adjacency list.

Problem 6 (20 points)**.** *In class, we have discussed how to use DFS for topological sorting of a DAG. Someone tries to come up with something new by trying to use BFS for topological sorting of a DAG. Below is his/her code. Does his/her code work ? If yes, explain why ? If no, give a counter example.*

```
time = 0; //global clock

//G: the graph. s: the node to start with
graph_BFS(G,s)
{
	time = time + 1; //New operation
	s.start_time = time; //New operation
	s.visited = true;
	FIFO.enqueue(s);

	while(FIFO.size > 0)
		u = FIFO.dequeue();
		print(u);
		time = time + 1; //New operation
		u.end_time = time; //New operation

		for every (u,v) \in E
			if v.visited == false
			v.visited = true;
			time = time + 1; //New operation
			v.start_time = time; //New operation
			FIFO.enqueue(v);
}
```

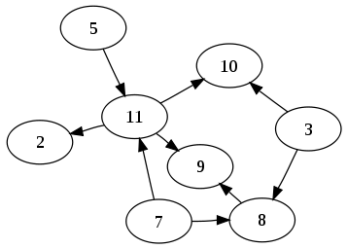
CSCD320 Homework6, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 25, 2014 (Sunday)

```
BFS(G) {
       for each vertex u \in G.V
              u.visited = false;
       for each u \in G.V
              if u.visited = false
                     graph_BFS(G,u)
}

topological_sort(G)
{
       1. call BFS(G) to compute finishing time u.end_time for each vertex u
       2. as each vertex is finished, insert it onto the *END* of a linked list
       3. return the linked list of vertices
}
```

Counter Example:



According the above algorithm and the shown DAG picture

Do the first BFS starting with 5 and the linked list will be: 5 → 11 → 9 → 2 → 10

Next pick 7 out of the remaining untouched nodes to start the next BFS, and only search nodes that can be touched from 7 and haven't been searched by the earlier BFS. This will give 7 → 8. Add this to the end of the linked list and the result will be : 5 → 11 → 9 → 2 → 10 →7 →8

Lastly, BFS is called again on the only remaining un-touched node 3, and it will finish immediately and be added to the end of the linked list to give the final list:

5 → 11 → 9 → 2 → 10 →7 →8 → 3

Clearly this did not sort in topological order. In the original DAG, 10 is in 3's path and therefore 3 should be before 10 in the linked list, but it is not. 9 is in 8's path and therefore 8 should be before 9 in the list, but it is not. 11 and all that connect from it are in 7's path, and therefore 7 should be in the list before 11 but it is not. Therefore, this code does not work.