Name & UNI: _Amy Qi_____ Date: _Dec 4_____

Point values are assigned for each question. Points earned: ____ / 100, = ____ %

1. Joe Pancake has just invented what he thinks is a great sorting algorithm. Consider the following code:

```
/**
 * Returns index of the maximum element in array[0..n-1].
 */
int find_max(int array[], const int length) {
   int max_index, i;
   for (max_index = 0, i = 0; i < length; ++i) {
       if (array[i] > array[max_index]) {
           max_index = i;
       }
   }
   return max_index;
}

/**
 * Reverses array[0..limit].
 */
void flip(int array[], int limit) {
    int temp, start = 0;
    while (start < limit) {
        temp = array[start];
        array[start] = array[limit];
        array[limit] = temp;
        start++;
        limit--;
    }
}

void pancake_sort(int array[], const int length) {
    for (int curr_size = length; curr_size > 1; --curr_size) {
        int max_index = find_max(array, curr_size);
        if (max_index != curr_size-1)  {
            flip(array, max_index);
            flip(array, curr_size-1);
        }
    }
}
```

Show the array [2, 4, 1, 5, 3] after the for loop in pancake_sort executes once.
(5 points, one for each element in the correct position)

[3,2,4,1,5]

What is the best-case complexity of the pancake_sort algorithm above including helper functions?
(1 point for symbol, 1 for function)

Theta(n^2)

What is the worst-case complexity of the pancake_sort algorithm above including helper functions?
(1 point for symbol, 1 for function)

Theta(n^2)

Which of the elementary sorting algorithms discussed in class is *closest to (but not exactly the same as)* what Joe has written? Use the complexities to guide your response. Circle your answer. (1 point)

bubble / selection / insertion

2. You are given a collection of bolts of different widths and corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut.
Note: You *cannot* get credit for parts b and c if your answer to part a is incorrect.
   a) Design an algorithm for this problem with *efficient* average-case complexity faster than quadratic run time. (6 points)

Pick a pivot from the array of nuts. Based on this pivot, divide the array of bolts into two halves where one collection of bolts is smaller than the bolt and the other is larger. This way we can also find the bolt that pairs with the pivot nut.
Now use the bolt that matches the nut to divide the array of nuts into two just like what we did with the bolts.
Now we have the array of nuts and bolts like below.
[LHS: nuts smaller than the pivot nut] pivot nut [RHS: nuts larger than the pivot nut]
[LHS: bolts smaller than the pivot bolt] pivot bolt [RHS: bolts larger than the pivot bolt]
Carry out the process described above with the LHS of pivot nut and the LHS of pivot bolt, i.e. choose a pivot nut, divide the bolts, then divide the nuts based on the pivot bolt found.
Carry out the process described above with the RHS of pivot nut and the RHS of pivot bolt.

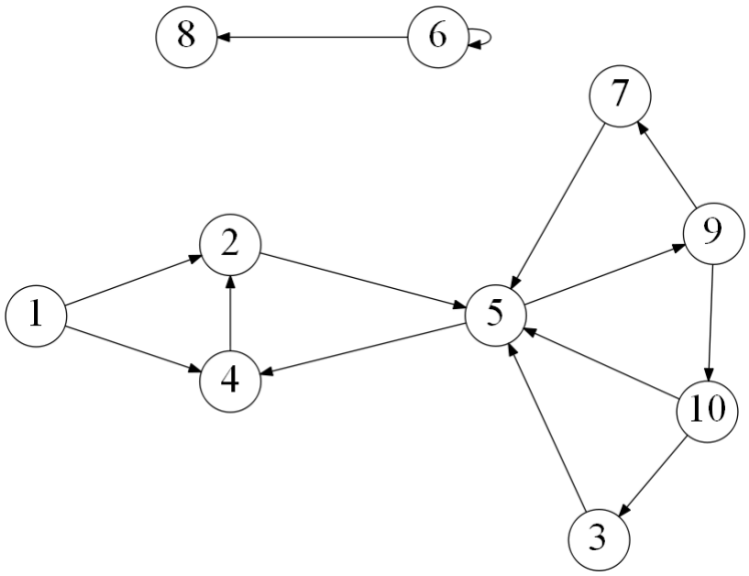
Repeat the entire process until all nuts and bolts are matched.

   b) What is the average-case complexity of your algorithm? (1 point for symbol, 1 for function)

Theta(nlogn)

c) What algorithm did we study this semester that is most similar to the algorithm you've designed? (2 points)
Quicksort

3. List the order in which the vertices are visited with a breadth-first search starting with vertex 1. If there are multiple vertices adjacent to a given vertex, visit the adjacent vertex with the lowest value first. (10 points)



1, 2, 4, 5, 9, 7, 10, 3, 6, 8

How the sequence is found:

Start with the first unmarked vertex, which is 1, call bfs(1)

Mark and enqueue 1

Mark and enqueue 2 and 4

Dequeue 1

Mark and enqueue 5

Dequeue 2

Dequeue 4

Mark and enqueue 9

Dequeue 5

Mark and enqueue 7 and 10

Dequeue 9

Dequeue 7

Mark and enqueue 3

Dequeue 10

Dequeue 3

The first call to bfs( ) now returns, and we loop until an unmarked vertex, which is 6, and call bfs(6)

Mark and enqueue 6

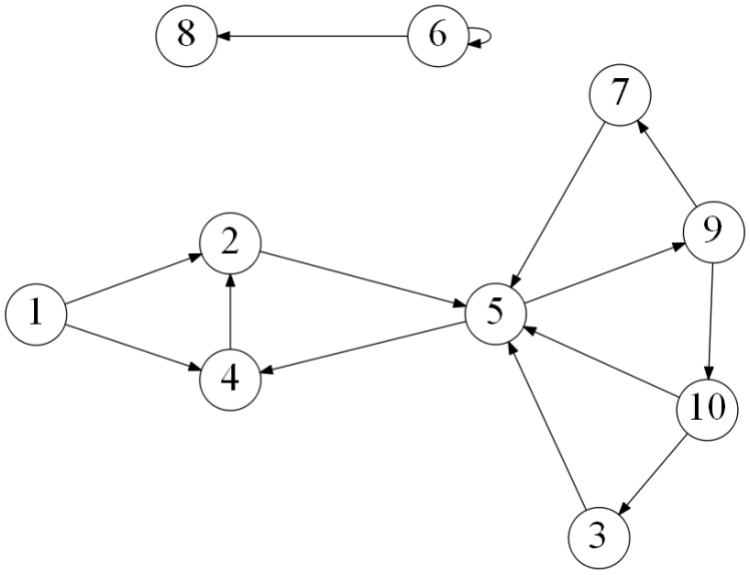
Mark and enqueue 8

Dequeue 6

Dequeue 8

The sequence of vertices marked is the sequence of vertices visited

4. Using the same graph as in the previous question, list the order in which the vertices are visited with a depth-first search starting with vertex 1. If there are multiple vertices adjacent to a given vertex, visit the adjacent vertex with the lowest value first. (10 points)



1, 2, 5, 4, 9, 7, 10, 3, 6, 8

How the sequence is found:

Start with the first unmarked vertex 1, call dfs(1), mark 1

Call dfs(2), mark 2

Call dfs(5), mark 5

Call dfs(4), mark 4, base case reached, back to the loop in dfs(5)

Call dfs(9), mark 9

Call dfs(7), mark 7, base case reached, back to the loop in dfs(9)

Call dfs(10), mark 10

Call dfs(3), mark 3, base case reached, roll all the way up, and dfs(1) returns

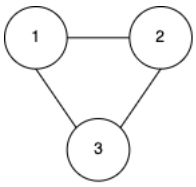
Now back in the main loop, the next unmarked node is 6, call dfs(6), mark 6

Call dfs(8), mark 8, base case reached, all the way up.

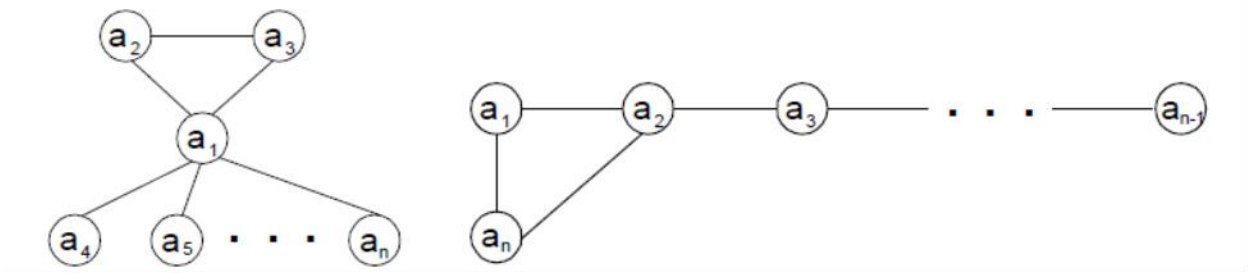
The sequence of nodes marked is the sequence of nodes visited.

5. On undirected graphs, does either of the two traversals, DFS or BFS, always find a cycle faster (i.e. in less operations) than the other? If yes, indicate which of them is better and explain why it is the case; if not, draw two graphs supporting your answer and explain the graphs. (10 points)

No. A counterexample is shown in the graph below.

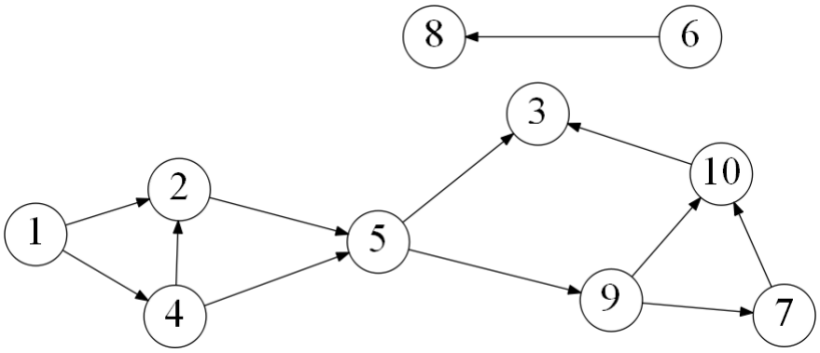


Suppose we start with 1, and visit the adjacent vertex with the lowest value first.
For BFS:
We visit vertex 1, vertex 2 and vertex 3, and learn that there is a cycle.
For DFS:
We visit vertex 1, vertex 2 and vertex 3, and learn that there is a cycle.
As demonstrated, there's no difference when it comes to the number of vertices visited. So there's no guarantee whether BFS or DFS is faster for cycle detection in undirected graph.



Also using this example.
If we no longer persist the order to visit adjacent nodes, a possible sequence of nodes traversed for the graph on the left hand side using BFS would be a1, a4, a5…an, a2, a3, where all nodes need to be visited before the cycle is detected. Yet for DFS, a possible sequence of nodes visited is a1, a2, a3, a4…an, where the cycle is detected right after the traversal started. In this case, DFS is faster than BFS.
For the graph on the right hand side, however, a possible sequence of nodes visited with DFS could be a1,a2,….an-1, an, whereas a possible sequence for BFS could be a1, a2, an, a3…an-1. This time BFS is definitely faster than DFS because BFS detects the cycle within visiting the first three nodes, yet DFS needs to walk through all nodes.
Overall, there is no guarantee which method is faster.

6. Consider the following graph:



List the order in which the vertices are visited with a topological sort, as shown in class. Always process the vertex with the lowest number first if several vertices have indegree 0. (10 points)

1,4,2,5,6,8,9,7,10,3

How the sequence is found:
L=
S= 1,6

L=1
S=6,4

L=1,4
S=6,2

L=1,4,2
S=6,5

L=1,4,2,5
S=6,9

L=1,4,2,5,6
S=9,8

L=1,4,2,5,6,8
S=9

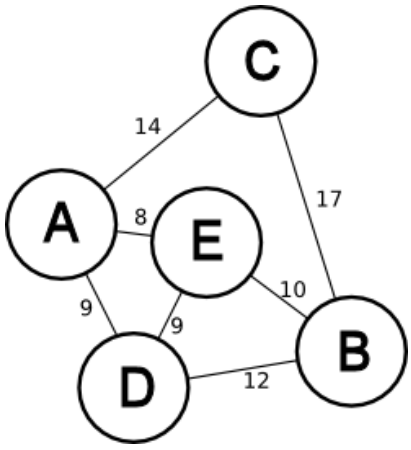
L=1,4,2,5,6,8,9
S=7

L=1,4,2,5,6,8,9,7
S=10

L=1,4,2,5,6,8,9,7,10
S=3

L=1,4,2,5,6,8,9,7,10,3
S=

7. Consider the following graph:



Apply Kruskal's algorithm to find the minimum spanning tree. Edges are sorted first by length, and in the event of a tie, by name, where the two letters are in alphabetical order. Use `makeset(x)`, `find(x)`, and `union(x, y)` to determine if there are cycles.

a) Circle the edges that are part of the minimum spanning tree. (4 points)
AC, AD, AE, BC, BD, BE, DE

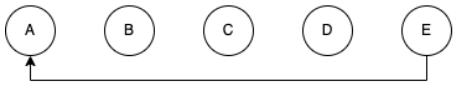
Sort all the edges and decide if each one of them is in the set of edges that form a MST
AE (add), AD (add), DE (skip), BE (add), BD (skip), AC (add), BC (skip)

b) What is the weight of the minimum spanning tree? (1 point)
The weight is AC + AD + AE + BE = 8+9+14+10 = 41

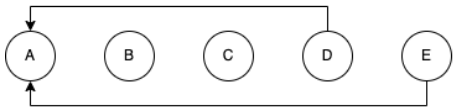
c) Draw the tree that results from applying the union-find algorithm for cycle detection. When drawing the tree, put vertices with lower letters in the left subtrees so that all the vertices in a level are sorted alphabetically from left to right. (5 points)

makeset

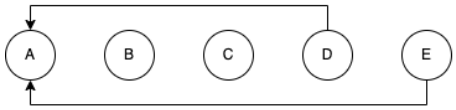
AE find(A) = A find(E) = E, union(A,E)



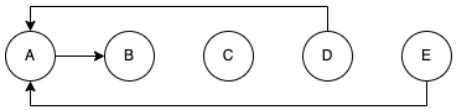
AD find(A) = A find(D) = D, union(A,D)



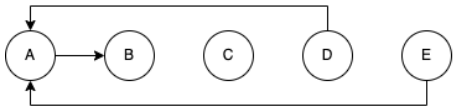
DE find(D) = A, find(E) = A skip



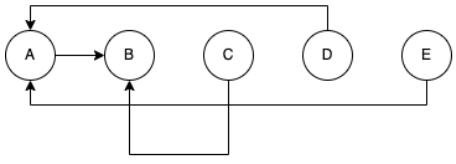
BE find(B) = B, find(E) = A, union(B,A)



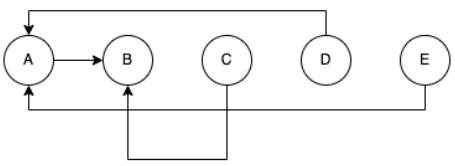
BD find(B) = B, find(E) = B skip



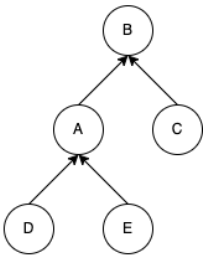
AC find(A) = B, find(C) = C, union(A,C) typo: should be union(B,C) here



BC find(B) = B, find(C) = B skip



final result

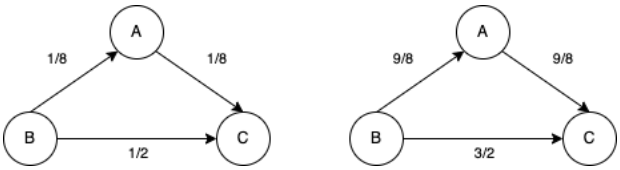


8. Using the same graph as in the previous question, use Prim's algorithm to find the minimum spanning tree starting with vertex A. Fill in the table below, as shown in lecture. (10 points)

Tree vertices	Remaining vertices
A(-,-)	B(-, inf), C(A,14), D(A,9), E(A,8)
E(A,8)	B(E,10), C(A,14), D(A,9)
D(A,9)	B(E,10), C(A,14)
B(E,10)	**C(A,14)**
C(A,14)	

9. In a weighted graph, assume that the shortest path from a source 'S' to a destination 'T' is correctly calculated using Dijkstra's shortest path algorithm. If we increase weight of every edge by 1, does the shortest path always remain the same? If so, prove it. If not, give a counterexample. (10 points)

If we take into consideration fractions, this statement is not true. A counter example is shown below.



Suppose we want to find the shortest path from B to C.

Since we don't have negative weight edges in either diagram, Dijkstra will always yield the correct solution for the shortest path between B and C.

However, the solution differs for the two diagram.

Before increasing the weights, the solution derived from Dijkstra is B->A->C since 1/8+1/8 = ¼ < ½

After increasing the weights, the solution derived from Dijkstra will become B->C because 9/8 + 9/8 = 9/4 > 3/2

Therefore the shortest path changed.

10. Suppose you have coins in a row with the following monetary value in cents: 7 1 3 2 1 7 5
You want to take coins such that you maximize the value of collection subject to the constraint that no two adjacent coins may be taken. Use dynamic programming to fill in the table. (7 points)

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
C	**0**	**7**	**1**	**3**	**2**	**1**	**7**	**5**
F	**0**	**7**	**7**	**10**	**10**	**11**	**17**	**17**
S	**0**	**0**	**1**	**1**	**3**	**3**	**4**	**6**

What is the maximum amount of money you can take? (1 point)
The maximum amount of money that can be taken is 17.

What coins did you take? List the indices of the coins taken from lowest to highest, assuming that first 7-cent coin is in index 1. (2 points)
Indexes are: 1,3,6