

**Spring 2021 CSE3080 Data Structures**  
**Final Exam**

※ Write your answers on the answer sheet. Make sure your answers are clearly recognizable.

**1. Graph Definitions - (1)-(8) 1pt - Total 8 pts**

(1) A ( ) graph is a graph whose edge is represented by a pair  $\langle u, v \rangle$ ;  $u$  is the tail and  $v$  is the head of the edge. The edge  $\langle u, v \rangle$  is drawn as an arrow from  $u$  to  $v$ .

(2) If there is an edge between every vertex pair, the graph is called a ( ) graph.

(3) A ( ) from vertex  $u$  to vertex  $v$  in graph  $G$  is a sequence of vertices  $u, i_1, i_2, \dots, i_k, v$  such that  $(u, i_1), (i_1, i_2), \dots, (i_k, v)$  are edges in  $G$ .

(4) A ( ) is a **(3)** in which all vertices except possibly the first and the last are distinct.

(5) A ( ) is a **(4)** in which the first and the last vertices are the same.

(6) A ( ) of an undirected graph  $G$  is a maximal connected subgraph of  $G$ .

(7) A ( ) is a connected acyclic graph.

(8) A ( ) of a vertex is the number of edges incident to that vertex.

## 2. Graph Representations - (1)-(4) 2pts - Total 8 pts

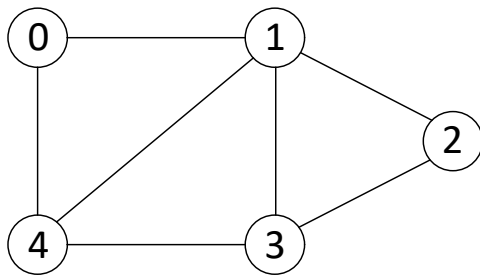
(1) Represent graph A using an **adjacency list**. The adjacency list should be represented as an array of linked lists. In a linked list, the vertices must be sorted in the ascending order of the vertex ID. (Draw the adjacency list.)

(2) Represent graph A using an **adjacency matrix**. (Draw the adjacency matrix.)

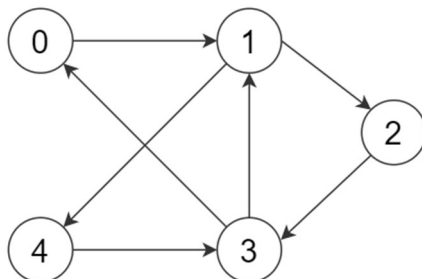
(3) Represent graph B using an **adjacency list**. The adjacency list should be represented as an array of linked lists. In a linked list, the vertices must be sorted in the ascending order of the vertex ID. (Draw the adjacency list.)

(4) Represent graph B using an **adjacency matrix**. (Draw the adjacency matrix.)

**Graph A**



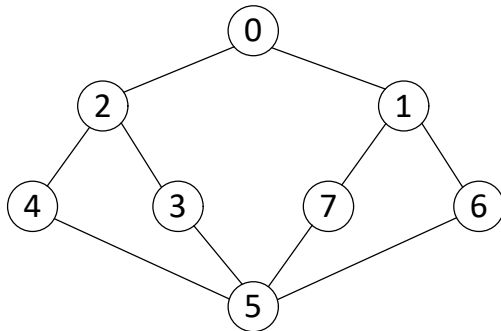
**Graph B**



### 3. Graph Search - (1)-(2) 4pts - Total 8 pts

(1) We are going to do a depth-first search (DFS) on **Graph C** starting from vertex 0. Write the vertex IDs in the order they are visited. When there are multiple candidate vertices to visit next, we choose the one with the lower vertex ID. (e.g., If both vertex 1 and vertex 2 can be the next node to visit, we visit vertex 1.)

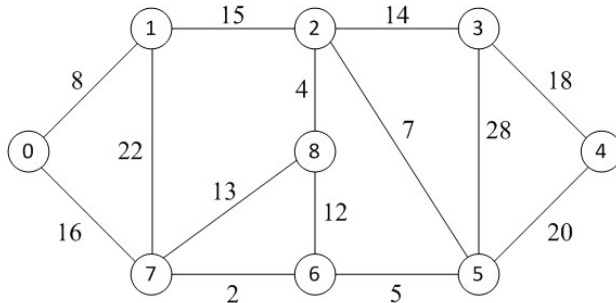
**Graph C**



(2) We are going to do a breadth-first search (BFS) on **Graph C** starting from vertex 0. Write the vertex IDs in the order they are visited. When there are multiple candidate vertices to visit next, we choose the one with the lower vertex ID. (e.g., If both vertex 1 and vertex 2 can be the next node to visit, we visit vertex 1.)

#### 4. Minimum Spanning Trees - (1)-(2) 6pts, (3) 3pts - Total 15 pts

**Graph D**



(1) For **Graph D**, find the minimum spanning tree using **Prim's algorithm**, starting from vertex 0. Write the order of edges included in the minimum spanning tree. You can write the edge between vertex **i** and **j** as **(i, j)**.

(2) For **Graph D**, find the minimum spanning tree using **Kruskal's algorithm**. Write the order of edges included in the minimum spanning tree.

(3) Below is the pseudocode for implementing Kruskal's algorithm using disjoint sets. **MAKE-SET(v)** creates a graph with a single node representing vertex **v**. **FIND-SET(v)** finds root of the tree where vertex **v** belongs. **UNION(u, v)** combines two trees that have **u** and **v** in them. For the UNION operation, we use the **weightedUnion**, where root of the smaller tree becomes child of root of the larger tree. For ordering edges by weight, we use a minimum heap.

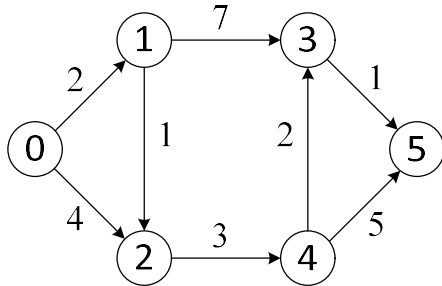
If the number of vertices is  $|V|$  and the number of edges is  $|E|$ , what is the worst-case time complexity of the algorithm? Use the Big-O notation.

```
KRUSKAL(G) :
1  A = ∅
2  foreach v ∈ G.V:
3      MAKE-SET(v)
4  foreach (u, v) in G.E ordered by weight(u, v), increasing:
5      if FIND-SET(u) ≠ FIND-SET(v) :
6          A = A ∪ {(u, v)}
7          UNION(u, v)
8  return
```

### 5. Single Source Shortest Paths (1)-(5) 3pts - Total 15 pts

For **Graph E**, we would like to find the shortest path from vertex 0 to all other vertices using Dijkstra's algorithm. Initially (Step 0), the set **SPT** includes vertex 0, and the **distance table** is as given below.

**Graph E**



Step 0: SPT = { 0 }

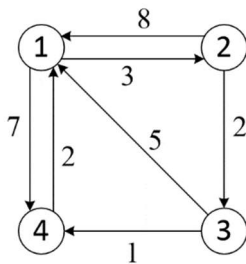
vertex	distance
1	2
2	4
3	INF
4	INF
5	INF

- (1) Write SPT and distance table after Step 1.
- (2) Write SPT and distance table after Step 2.
- (3) Write SPT and distance table after Step 3.
- (4) Write SPT and distance table after Step 4.
- (5) Write SPT and distance table after Step 5.

## 6. All-Pairs Shortest Paths (1)-(4) 4pts - Total 16 pts

For **Graph F**, we would like to find the shortest path between all pairs of vertices. We define a matrix  $A^k[i][j]$ , that represents the length of shortest paths from a source vertex to a destination vertex.  $A^k[i][j]$  is defined as the shortest path length from vertex  $i$  to vertex  $j$ , only taking vertices 1 through  $k$  as intermediate vertices.

### Graph F



Initially (Step 0),  $A^0$  matrix looks like the following.

	1	2	3	4
1	0	3	inf	7
2	8	0	2	inf
3	5	inf	0	1
4	2	inf	inf	0

- (1) Write  $A^1$  after adding vertex 1 as an intermediate vertex.
- (2) Write  $A^2$  after adding vertex 2 as an intermediate vertex.
- (3) Write  $A^3$  after adding vertex 3 as an intermediate vertex.
- (4) Write  $A^4$  after adding vertex 4 as an intermediate vertex.

## 7. Sorting (1)-(2) 5pts - Total 10 pts

Following is an example code implementing Quick Sort.

```
#define SWAP(a,b) {int tmp; tmp = a; a = b; b = tmp;}

void quicksort(int A[], int left, int right) {
    int pivot;
    if(right - left > 0) {
        pivot = partition(A, left, right);
        quicksort(A, left, pivot-1);
        quicksort(A, pivot+1, right);
    }
}

int partition(int A[], int left, int right) {
    int i, pivot;
    pivot = left;
    for(i = left; i < right; i++) {
        if(A[i] < A[right]) {
            SWAP(A[i], A[pivot]);
            pivot++;
        }
    }
    SWAP(A[right], A[pivot]);
    return pivot;
}
```

(1) Suppose the input array A looks like the following:

26	5	77	1	61	11	59	15	48	19
----	---	----	---	----	----	----	----	----	----

What is the return value of function partition when A is the array above, left is 0 and right is 9? In other words, what is the return value of `partition(A, 0, 9)`?

(2) What does array A look like after the first call to function partition? We assume that the input array is the same as in problem (1). Describe the contents of the array.

## 8. Heaps (1)-(2) 5pts - Total 10 pts

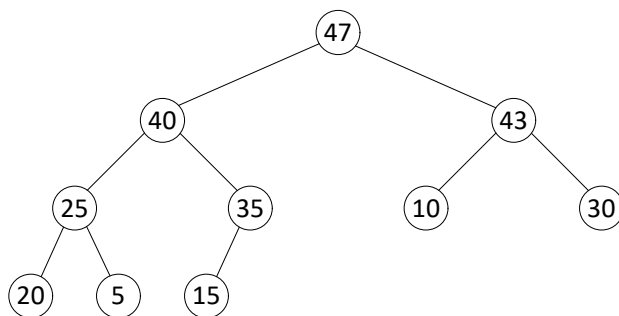
A max heap is a complete binary tree that is a max tree. A max tree is a tree in which the key value of each node is no smaller than the key values in its children (if any.)

We can implement the max heap using an array. The array `heap[]` looks like the following:

- `heap[0]` is not used.
- `heap[1]` is the root of the max heap.
- The children of `heap[i]` are `heap[2*i]` and `heap[2*i+1]` if they exist.

For insertion and deletion on the heap, we use the code shown in **Appendix A** (the last page).

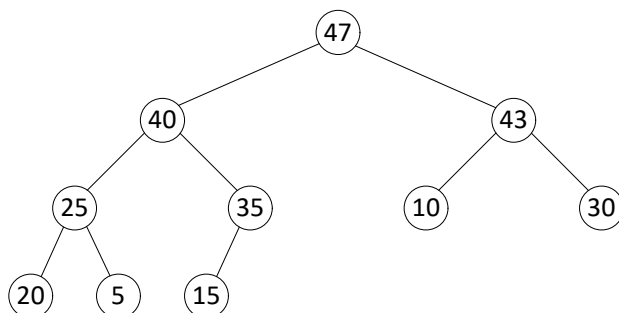
(1) In the heap shown below, we insert a node that has key **50**. Draw the heap structure after the node is inserted.



heap 

	47	40	43	25	35	10	30	20	5	15					
--	----	----	----	----	----	----	----	----	---	----	--	--	--	--	--

(2) In the heap shown below, we delete a node that has the maximum key. Draw the heap structure after the node is deleted.



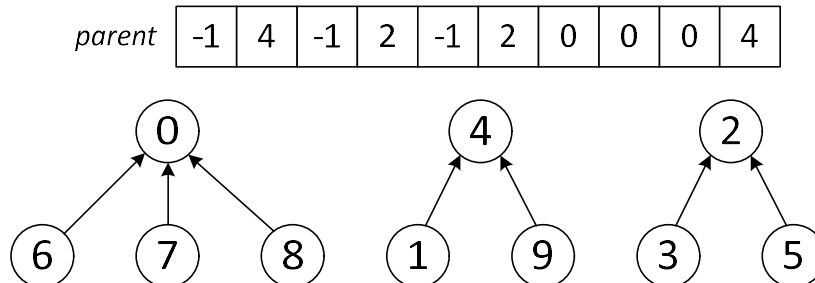
heap 

	47	40	43	25	35	10	30	20	5	15					
--	----	----	----	----	----	----	----	----	---	----	--	--	--	--	--



### 9. Disjoint Sets (1)-(2) 2pts (3) 6pts - Total 10 pts

We can represent disjoint sets using an array. Suppose we have nodes that are numbered from 0 to  $n-1$ . All we need is an array named *parent*, where the value of *parent*[*i*] is ID of the parent node of node *i*. If node *i* is a root node, *parent*[*i*] is -1. Below is an example disjoint sets and their array representation.



Two basic operations are defined on disjoint sets: Find(*i*) and Union(*i*, *j*). A simplest form of the two functions are shown below.

```
int simpleFind(int i) {  
    for( ; parent[i] >= 0; i = parent[i])  
        ;  
    return i;  
}  
  
void simpleUnion(int i, int j) { /* i and j must be roots */  
    parent[i] = j;  
}
```

(1) Suppose we start with initial disjoint sets of **n** nodes like the following. Then, we consecutively call **simpleUnion** on arbitrary two trees until eventually the sets become a **single tree**.



What is the maximum height of **the resulting single tree**? Use the following definition on height of a tree.

※ *Height of a tree* – The height of a tree is the number of edges on the longest downward path between the root and a leaf. If the tree has a single node (only the root node), its height is zero.

(2) What is the worst-case time complexity of **simpleFind**, when function **simpleUnion** is used for union operation on disjoint sets with **n** nodes? Use the Big-O notation.

(3) In order to improve the time complexity of the find operation, we design **weightedUnion** function which balances the tree height. The **weightedUnion** function does the following.

- The parent of a root node is no longer -1. Instead, it is -1 times the number of nodes in the tree. So, the parent values in the example given above is changed as the following.

parent	-4	4	-3	2	-3	2	0	0	0	4
--------	----	---	----	---	----	---	---	---	---	---

- When combining two trees using **weightedUnion(i, j)**, the root of the smaller tree becomes a child of the root of a larger tree. (The size of a tree is the number of nodes in the tree.) If the size of two trees is the same, j becomes a child of i.

Write the code to complete the **weightedUnion** function below. Assume `parent[]` is declared as a global variable.

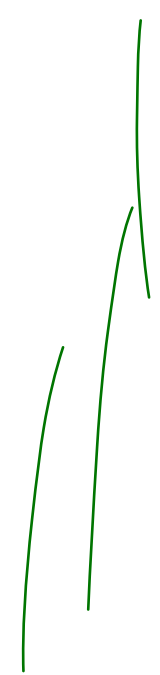
```
void weightedUnion(int i, int j) { /* i and j must be roots */
    int temp;

    /* maximum number of semicolons: 5 */

}
```

- End of the Exam -

Great work! Thanks for taking the course.



## Appendix A: functions used in Problem 8.

```
void insert_max_heap(element item, int *n) {
    /* insert item into a max heap of current size *n */
    int i;
    if(HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full.\n");
        exit(1);
    }
    i = ++(*n);
    while((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```

```
element delete_max_heap(int *n) {
    /* delete element with the highest key from the heap */
    int parent, child;
    element item, temp;
    if(HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty");
        exit(1);
    }
    /* save value of the element with the largest key */
    item = heap[1];
    /* use the last element in the heap to adjust heap */
    temp = heap[(*n)--];
    parent = 1;
    child = 2;
    while(child <= *n) {
        /* find the larger child of the current parent */
        if((child < *n) && (heap[child].key < heap[child+1].key)) child++;
        if(temp.key >= heap[child].key) break;
        /* move to the next lower level */
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return item;
}
```

