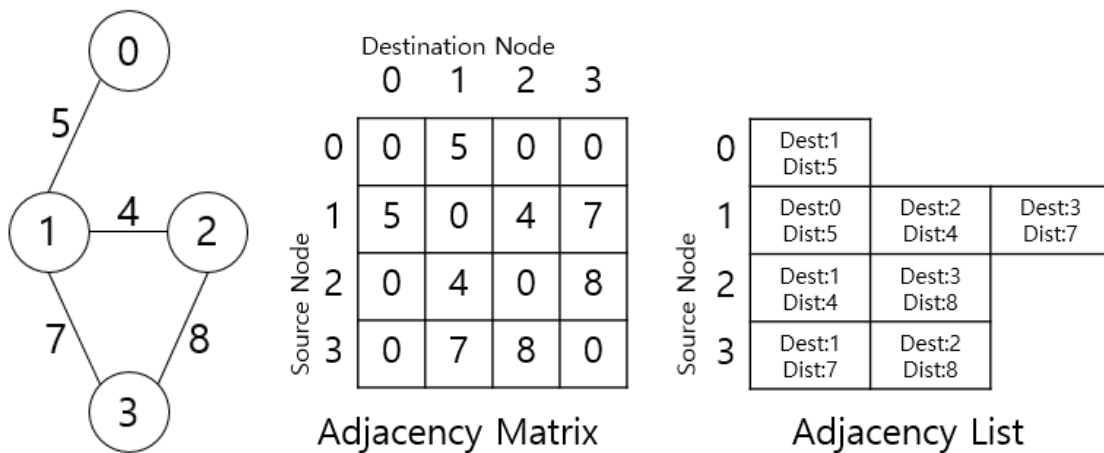


## Homework #4

Nov. 27<sup>th</sup> (Wed.) ~ Dec. 11<sup>th</sup> (Wed.) 23:59

\* Be sure to read the note on the last page.

**P1 (30pt). [Adjacency Matrix/List]** There are two ways to express a graph consisting of nodes and edges connecting nodes. One is adjacency matrix and the other is adjacency list. When the number of nodes in the graph is  $v$ , the size of adjacency matrix is  $v \times v$ , and the  $(i, j)^{\text{th}}$  element of the matrix represents the weight (or distance) of the edge connecting node  $i$  and node  $j$ . The adjacency list, on the other hand, consists of  $v$  linked list nodes, which store information from the node  $i$  and its neighboring nodes. In this problem, implement the adjacency list using dynamic array instead of linked list.



A. (10pt) Implement the constructor and the member functions of the Edge class when the class is defined as follows to represent each edge in the adjacency list.

Files to submit : HW4\_Edge.cpp (※ Don't include main function in .cpp file)

#### Edge Class

- **Definition**
  - The class representing weighted edge
- **Member Variable** (define as **Private**)
  - **int idx** : the index of the edge
  - **int src, int dest** : the source and destination node of the edge
  - **int dist** : the weight (or distance) of the edge
- **Constructor** (define as **Public**)
  - **Edge()** : initialize *idx, src, dest, dist* to 0
  - **Edge(int i, int s, int d, int w)** : initialize *idx, src, dest, dist* to *i, s, d, w* respectively
- **Member Function** (define as **Public**)
  - **int getIdx(), int getSrc(), int getDest(), int getDist()** : return *idx, src, dest, dist*
  - **void setIdx(int), void setSrc(int), void setDest(int), void setDist(int)** :  

*set idx, src, dest, dist*
  - **void print()** : print information of the edge ("*idx* *src* – *dest* (*dist*)")
  - **bool operator <(const Edge& e)** : return true if the *dist* of this edge is less than that of edge *e*
  - **bool operator >(const Edge& e)** : return true if the *dist* of this edge is greater than that of edge *e*
  - **Edge& operator=(const Edge& e)** : copy data of edge *e*

**B. (10pt) Implement the constructor and the member functions of the Node class when the class is defined as follows to represent each node in the adjacency list.**

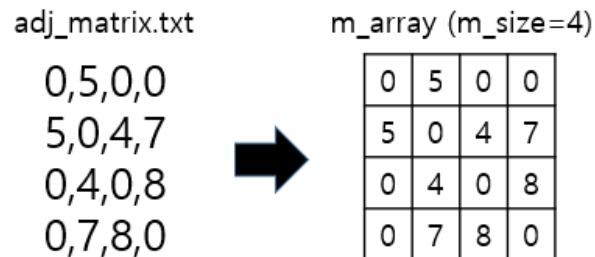
**(Node class is used to implement Dijkstra algorithm in P2)**

**Files to submit : HW4\_Node.cpp (※ Don't include main function in .cpp file)**

#### Node Class

- **Definition**
  - The class representing a node of a graph
- **Member Variable** (define as **Private**)
  - **int idx** : the index of the node
  - **int dist** : distance from the source node
- **Constructor** (define as **Public**)
  - **Node()** : initialize *idx*, *dist* to 0
  - **Node(int i, int d)** : initialize *idx*, *dist* to *i*, *d* respectively
- **Member Function** (define as **Public**)
  - **int getIdx(), int getDist()** : return *idx*, *dist*
  - **void setIdx(int), void setDist(int)** : set *idx*, *dist*
  - **void print()** : print information of the node ("*idx* (*dist*)")
  - **bool operator <(const Node& n)** : return true if the *dist* of this node is less than that of the node *n*
  - **bool operator >(const Node& n)** : return true if the *dist* of this node is greater than that of the node *n*
  - **Node& operator=(const Node& n)** : copy data of *n*

- C. (10pt) The given `adjReader()` function uses the given `tokenize()` function to convert the comma(,)-separated matrix into 2D int array by reading the text file as follows.



Implement the `matrixToList()` functions that takes the adjacency matrix and the number of nodes and creates an adjacency list in the form of a dynamic array. *n\_edge* represents the number of edges connected to each node. For example, *n\_edge* is [1, 3, 2, 2] in the figure. `adjList` is a dynamic 2D Edge array. `adjList[i][j]` represents that the *j*-th (*j* ≥ 0) Edge object connected to node *i*. For example, `adjList[1][1]` is an Edge object whose *idx* is 2, *src* is 1, *dest* is 2 and *dist* is 4.

Files to submit : HW4\_Adj.cpp (※ Don't include main function in .cpp file)

When HW4\_Adj\_Test.cpp is executed with `adj_matrix1.txt`, the result is as follows.

```

C:\Windows\system32\cmd.exe
Adjacency Matrix :
0 5 0 0
5 0 4 7
0 4 0 8
0 7 8 0

Adjacency List :
[0] 0 - 1 <5>
[1] 1 - 0 <5>    [2] 1 - 2 <4>    [3] 1 - 3 <7>
[4] 2 - 1 <4>    [5] 2 - 3 <8>
[6] 3 - 1 <7>    [7] 3 - 2 <8>
계속하려면 아무 키나 누르십시오 . . .

```

**P2 (35pt). [Dijkstra Algorithm] Single-source shortest path problem is defined as follows.**

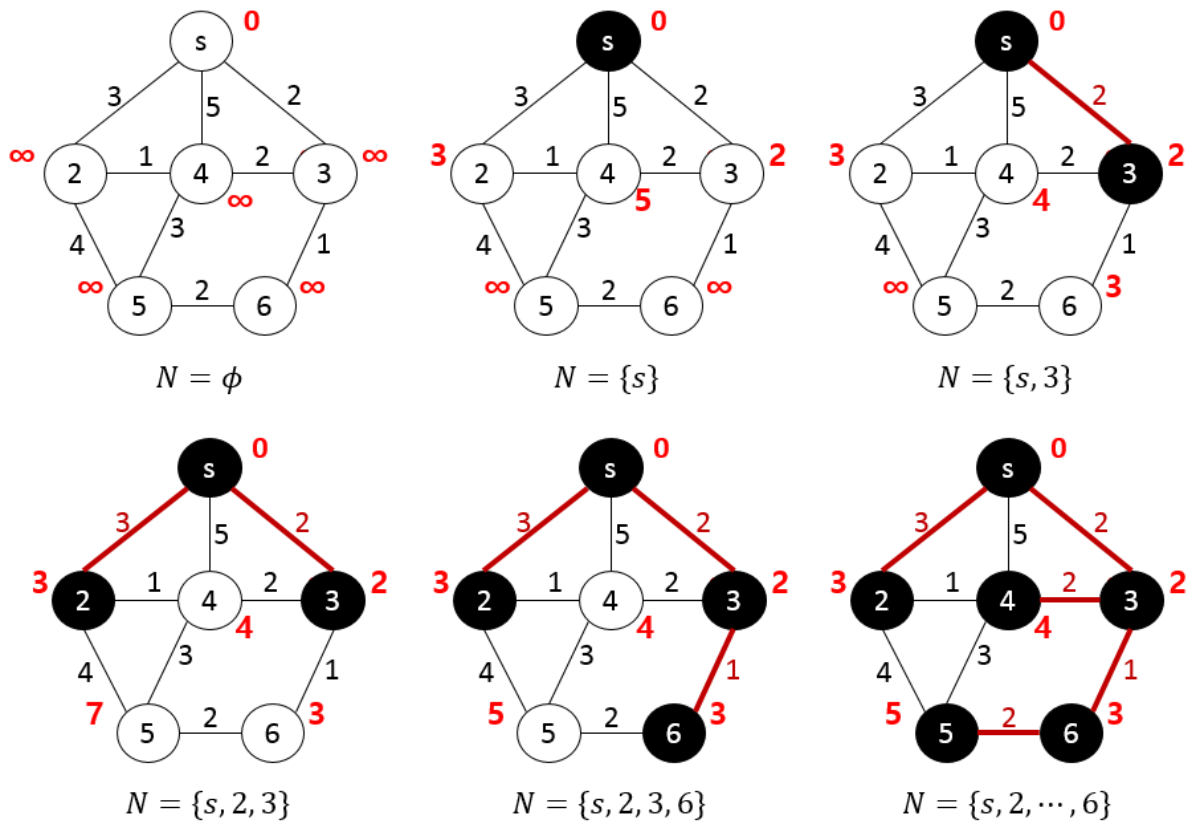
**Single-source shortest path problem**

- **Definition**
  - Edge weighted graph  $G = (V, E, w)$  ( $V$  : Vertex,  $E$  : Edge,  $w$  : Weight of edge)
  - All weights of edges are not negative
  - The problem to find the shortest path from a node  $v$  in  $G$  to all vertices in  $V$

**To solve this problem, Dijkstra algorithm is used.**

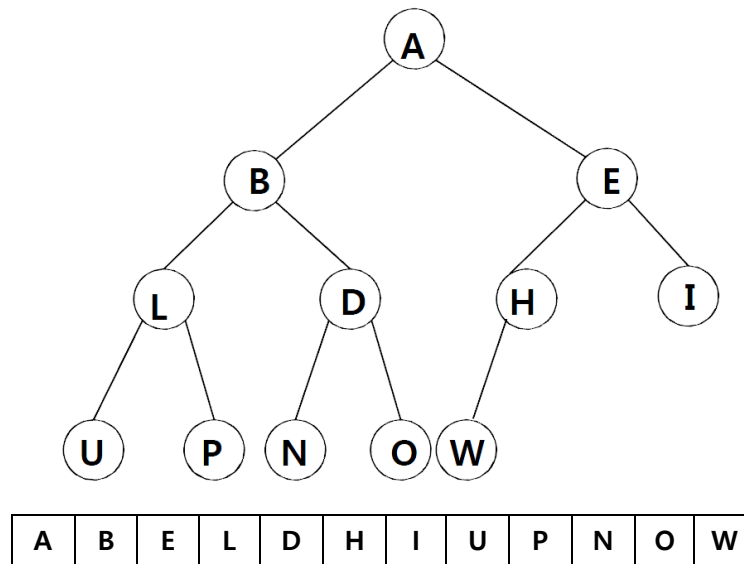
**Dijkstra algorithm**

- **Notation**
  - $N$  : set of nodes whose shortest path from source node  $s$  has already been computed
  - $T$  : set of edges that forms the shortest path
  - $D_j$  : current distance from source node  $s$  to node  $j$
- **Initialization** : Start from source node  $s$ 
  - $N = \{s\}$ ,  $T = \phi$
  - For all node  $j$ ,  $D_j = w_{sj}$  ( $w_{sj}$  : the weight of edge connecting node  $s$  and  $j$ )
- **Step A** : search for the next nearest node  $i$ 
  - For node  $j \notin N$ , find node  $i \notin N$  which satisfies  $D_i = \min D_j$  **(Use Heap Sort)**
  - Add node  $i$  to  $N$  and add edges connected with  $i$  to  $T$
  - Terminate when  $N = V$
- **Step B** : Update minimal cost **(Use Heap Sort & Adjacency List)**
  - For node  $j \notin N$  neighboring node  $i$ , update  $D_i = \min(D_i, D_j + w_{ij})$
  - Return to **step A**
- **Output** :  $D$



In the above figure, the algorithm starts from source node  $s$ . Initially, all  $D$  of the edges are initialized to  $INF(\infty)$ , and in the next step,  $D$  values of the node 2, 3, 4 which is connected to node  $s$  is updated respectively. Because the minimum value derived by heap sort is  $D_3 = 2$ , add node 3 to  $N$  and update  $D$  values of the node 4, 6 connected with node 3. This process is repeated until all nodes are added into  $N$ .

Heap sort is required to implement Dijkstra algorithm. Heap data structure is as follows; the parent node of  $i$ -th node is  $\left(\frac{i}{2}\right)$ -th node, left child is  $2i$ -th and right child is  $(2i + 1)$ -th node. (Note that the index of an array starts from 0, so it differs slightly.)



First, the `swap()` function exchanges  $a$ - and  $b$ -th element of the array. (i.e., when  $0 \leq a, b \leq (\text{Array size}-1)$ , exchange data of `array[a]` and `array[b]`)

The `maxHeapify()` functions is defined as follows.

#### maxHeapify

##### - Parameter

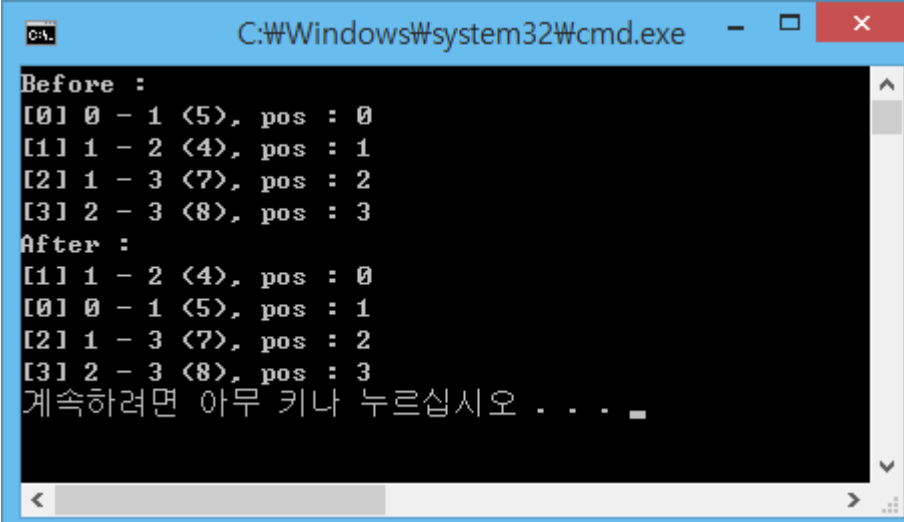
- **T\*** `arr` : Array containing Heap data ([A, B, E, ...] array in the figure)
- **int\*** `pos` : Array for indexing  
(e.g. `pos[0]` always represent the index of A, even if the position of the element is changed by the `swap()` function)
- **int** `size_heap` : the size of heap
- **int** `idx` : The index in array for current node

##### - Function

- Compare `arr[idx]` and its children and make the greatest value root node
- Finally, the root node of the heap should be the greatest value among all values in the heap
- Be able to use `swapPos()` and call `maxHeapify()` recursively

The `heapSort()` function uses `maxHeapify()` function to sort the values in ascending order. In this problem, `Edge` and `Node` class are used for template class `T`. Therefore, common functions such as `getIdx()`, `getDist()` are available. The final result of `heapSort()` is sorted from small to large.

When `HW4_Heap_Test.cpp` is executed with `adj_matrix1.txt`, the result is as follows.



```
C:\Windows\system32\cmd.exe
Before :
[0] 0 - 1 <5>, pos : 0
[1] 1 - 2 <4>, pos : 1
[2] 1 - 3 <7>, pos : 2
[3] 2 - 3 <8>, pos : 3
After :
[1] 1 - 2 <4>, pos : 0
[0] 0 - 1 <5>, pos : 1
[2] 1 - 3 <7>, pos : 2
[3] 2 - 3 <8>, pos : 3
계속하려면 아무 키나 누르십시오 . . .
```



Now, implement the Dijkstra algorithm using the Edge class, Node class, heap sort and adjacency list. The final result should print the shortest path from the specified source node to each node and edges contained in it. Use  $INF(10000)$  for  $INF(\infty)$  defined in HW4\_Dijkstra.h.

Files to submit : HW4\_Dijkstra.cpp (※ Don't include main function in .cpp file)

When HW4\_Dijkstra\_Test.cpp is executed with adj\_matrix3.txt, the result is as follows.

```

C:\Windows\system32\cmd.exe

Adjacency Matrix :
 0  4  0  0  0  0  0  8  0
 4  0  8  0  0  0  0 11  0
 0  8  0  7  0  4  0  0  2
 0  0  7  0  9 14  0  0  0
 0  0  0  9  0 10  0  0  0
 0  0  4 14 10  0  2  0  0
 0  0  0  0  0  2  0  1  6
 8 11  0  0  0  0  1  0  7
 0  0  2  0  0  0  6  7  0

Adjacency List :
[0] 0 - 1 <4>    [1] 0 - 7 <8>
[2] 1 - 0 <4>    [3] 1 - 2 <8>    [4] 1 - 7 <11>
[5] 2 - 1 <8>    [6] 2 - 3 <7>    [7] 2 - 5 <4>    [8] 2 - 8 <2>
[9] 3 - 2 <7>    [10] 3 - 4 <9>   [11] 3 - 5 <14>
[12] 4 - 3 <9>   [13] 4 - 5 <10>
[14] 5 - 2 <4>   [15] 5 - 3 <14>   [16] 5 - 4 <10>   [17] 5 - 6 <2>
[18] 6 - 5 <2>   [19] 6 - 7 <1>    [20] 6 - 8 <6>
[21] 7 - 0 <8>   [22] 7 - 1 <11>   [23] 7 - 6 <1>    [24] 7 - 8 <7>
[25] 8 - 2 <2>   [26] 8 - 6 <6>    [27] 8 - 7 <7>

Result :
D :
[0] <0>
[1] <4>
[2] <12>
[3] <19>
[4] <21>
[5] <11>
[6] <9>
[7] <8>
[8] <14>
계속하려면 아무 키나 누르십시오 . . .
  
```

**P3 (35pt). [Prim Algorithm] Minimum spanning tree is defined as follows.****Minimum spanning tree problem**

- **Definition**
  - Edge weighted graph  $G = (V, E, w)$  ( $V$  : Vertex,  $E$  : Edge,  $w$  : Weight of edge)
  - All weights of edges are not negative
  - The problem to find sub-tree of  $G$  whose total edge weight is minimum

To solve this problem, Prim algorithm is used.

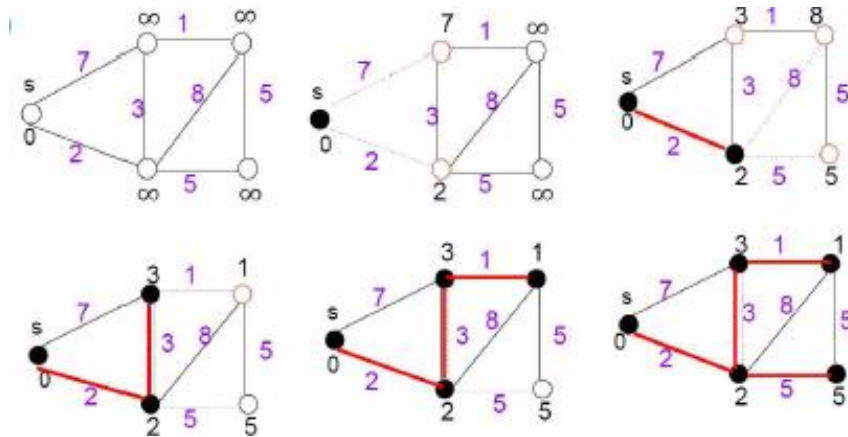
**Prim algorithm**

- **Notation**
  - $S$  : set of nodes in the spanning tree
  - $T$  : set of edges in the sub-tree whose total edge weight is minimum
  - $W_j$  : current distance from spanning tree to Node  $j$
- **Initialization** : Start from node  $s$ 
  - $S = \{s\}, T = \phi$
  - For all node  $j \neq s, W_j = \infty$   
( $W_j$  : the shortest distance among edges connecting spanning tree with Node  $j$ )
  - For all nodes  $j$  neighboring node  $s$ , update  $W_j = w_{sj}$   
( $w_{ij}$  : weight of edge connecting node  $i$  and  $j$ )
- **Step A** : search for the shortest safe edge
  - For node  $j \notin S$ , find node  $i \notin S$  which satisfies  $W_i = \min W_j$  **(Use Heap Sort)**
  - Add node  $i$  to  $S$  and add edges connected with  $i$  to  $T$
  - Terminate when  $S = V$
- **Step B** : update minimal cost **(Use Heap Sort & Adjacency List)**
  - For node  $j \notin S$  neighboring node  $i$ , update  $W_j = \min(W_j, w_{ij})$
  - Return to **Step A**
- **Output** :  $T$

※ If distances of nodes are the same, select a node arbitrarily among them.

※ Edge stored in T can be (0-1), (1-2) or (1-0), (2-1) (opposite).

※ Output should be in the order of edges added.



In the above figure, the algorithm starts from source node  $s$ . Initially, all  $W$  of the edge are initialized to  $INF(\infty)$ , and in the next step,  $W$  values of the node connected with node  $s$  are updated. Because the minimum value derived by Heap sort is 2, add node to  $S$  and update the  $W$  value for the connected node. This process is repeated until all nodes are added into  $S$ .

Now, implement the Prim algorithm using the Edge class, Node class, heap sort and adjacency list. The final result should print the edges of minimum spanning tree. Use  $INF(10000)$  for  $INF(\infty)$  defined in HW4\_Prim.h.

Files to submit : HW4\_Prim.cpp (※ Don't include main function in .cpp file)

When HW4\_Prim\_Test.cpp is executed with adj\_matrix2.txt, the result is as follows.

```

C:\Windows\system32\cmd.exe
Adjacency Matrix :
0  2  0  6  0
2  0  3  8  5
0  3  0  0  7
6  8  0  0  9
0  5  7  9  0

Adjacency List :
[0] 0 - 1 <2>    [1] 0 - 3 <6>
[2] 1 - 0 <2>    [3] 1 - 2 <3>    [4] 1 - 3 <8>    [5] 1 - 4 <5>
[6] 2 - 1 <3>    [7] 2 - 4 <7>
[8] 3 - 0 <6>    [9] 3 - 1 <8>    [10] 3 - 4 <9>
[11] 4 - 1 <5>  [12] 4 - 2 <7>  [13] 4 - 3 <9>

Result :
T :
0 - 1 <2>
1 - 2 <3>
1 - 4 <5>
0 - 3 <6>
계속하려면 아무 키나 누르십시오 . . .

```

### Note

- **Scoring will be based on execution results in Microsoft Visual Studio 2019.**
- **Do NOT modify given names of functions and variables! Otherwise score will be deducted.**
- **You can add member functions if needed. In this case, submit header file! Otherwise score will be deducted.**
- **Submit the files with the exact file names given! Otherwise score will be deducted.**
- **Output should be in the same form as the example given! Otherwise score will be deducted.**
- **Scoring will be done with more complex case than the example test case given.**
- **No plagiarism! If plagiarism is detected, 0 points will be given for the assignment and it will be notified to the professor.**
- **If you have any question, ask questions via e-mail only if they are not resolved after sufficient search has been done.**

#### ● How to submit

- Write code for each problem in each .h / .cpp file.
- Files should be saved **with exact file names** given in each problem.
- Compress your code into **one compressed file**.
  - ◆ The compressed file name should be "HW4\_(name)\_(student ID).zip" using zip compression.
  - ◆ e.g.) "HW4\_김태환\_2017-11111.zip"
- Submit the compressed file to the "Assignment 4" on the eTL course page.

#### ● Deadline for submission

- **By Wednesday, December 11<sup>th</sup>, 11:59 pm.**
- Submit the assignment **via e-mail within the deadline** if there is any problem when you submit it on eTL.  
E-mail : [ds@snucad.snu.ac.kr](mailto:ds@snucad.snu.ac.kr)
- **No delay submission (both eTL / e-mail). 0 points for late submission or no submission.**
- **Make sure that the file is attached and submitted! 0 points for submission without attachment.**