ID 2014311710 Name 김민우

# Information

Your code will be evaluated based on

* Completeness
* Readability
* Efficiency of algorithm.

# Dynamic Programming

1. You are given an n by m 2-dimensional matrix. You will find out a path with the longest length.

You can move to an adjacent right cell or an adjacent down cell only if the number in the next cell has difference in either +1 or -1 from the number in the current cell.

It is not necessary to the start and ending cells are (0, 0) and (n-1, m-1).

Example

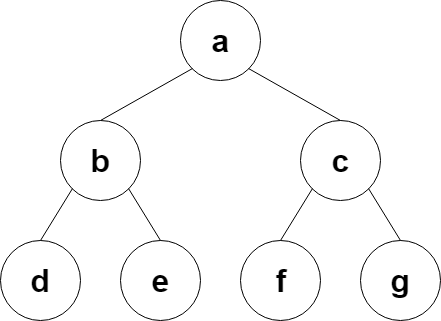
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 5 | 2 | 3 | 1 |
| 3 | 4 | 1 | 4 | 4 |
| 1 | 5 | 6 | 7 | 8 |
| 3 | 4 | 5 | 8 | 9 |
| 3 | 2 | 2 | 7 | 6 |

* 1. Write a Java program to find the longest path using the dynamic programming algorithm.
  2. What is the big-o complexity of your program?
  3. Write a Java program to find all longest paths if there are such multiple paths using the dynamic programming algorithm.

# 정답은 밑에 적어뒀습니다.

# Divide-N-Conquer

1. There are three depth-first tree traversals: inorder (left-parent-right), preorder (parent-left-right), and postorder (left-right-parent). You are given a full binary tree represented in inorder traversal. Convert this notation into preorder traversal.
2. Example



1. Inorder: {d, b, e, a, f, c, g}  
   Preorder: {a, b, d, e, c, f, g}
   1. Write a Java program using the divide-and-conquer algorithm.
   2. What is the big-o complexity of your program?
   3. Based on the code for a), improve the program so that it can also accept a complete (but not full) binary tree. Write a Java program and explain the modification briefly.

# 수업 Feedback (선택 사항)

답을 적으면 각각 2점을 받습니다. 답의 내용으로 불이익을 받지 않습니다.

* 1. 수업 중 좋았던 점들은 무엇입니까?
  2. 수업 중 아쉬웠던 점들은 무엇입니까?

# 제출

각각의 코드를 문제 파일에 삽입하세요.

문제 파일 이름을 학번\_한글이름.docx 로 변경한 후 iCampus에 제출하세요.

1-a)----------------------------------------------------------------------------------------------------------------

**package** path;

/\* Maximum cost path solution using dynamic programming \*/

**public** **class** MaxCostPath {

**private** **static** **final** **int** ***NN*** = 5;

**private** **static** **int**[ ][ ] *stor* = **new** **int** [***NN***][***NN***];

**private** **static** **int**[ ][ ] *path* = **new** **int** [***NN***][***NN***];

**private** **static** **int** *maxLen*=-1;

**private** **static** **int**[ ][ ] *maxPath* = **new** **int**[***NN***][***NN***];

**static** **void** minGrid( **int** cost[][], **int** mmm, **int** nnn ) {

//put cost in stor for our start

*stor*[mmm][nnn]=cost[mmm][nnn];

//fill in down DPs

**for**(**int** i=1; i<***NN***; i++){

**if**((cost[mmm][i]-cost[mmm][i-1]==1 || cost[mmm][i]-cost[mmm][i-1]==-1) &&

*stor*[mmm][i-1]!=0 ) {

*stor*[mmm][i]=cost[mmm][i]+*stor*[mmm][i-1];

}

}

//fill in right DPs

**for**(**int** j=1; j<***NN***; j++){

**if**((cost[j][nnn]-cost[j-1][nnn]==1 || cost[j][nnn]-cost[j-1][nnn]==-1) &&

*stor*[j-1][nnn]!=0 ) {

*stor*[j][nnn]=cost[j][nnn]+*stor*[j-1][nnn];

}

}

//fill in the rest of it

**for** (**int** k=1; k<***NN***; k++){

**for**(**int** l=1; l<***NN***; l++){

//according to cost values, if there are two candidates

**if** ((cost[k][l]-cost[k-1][l]==1 || cost[k][l]-cost[k-1][l]==-1)&&

(cost[k][l]-cost[k][l-1]==1 || cost[k][l]-cost[k][l-1]==-1)) {

//follow the larger stor value

**if**(*stor*[k-1][l]>=*stor*[k][l-1] && *stor*[k-1][l]!=0) {

*stor*[k][l]=cost[k][l]+*stor*[k-1][l];

}

**else** **if**(*stor*[k-1][l]<*stor*[k][l-1] && *stor*[k][l-1]!=0){

*stor*[k][l]=cost[k][l]+*stor*[k][l-1];

}

//according to cost values, if there is only down possible

} **else** **if**((cost[k][l]-cost[k-1][l]==1 || cost[k][l]-cost[k-1][l]==-1)

&& *stor*[k-1][l]!=0) {

*stor*[k][l]=cost[k][l]+*stor*[k-1][l];

//according to cost values, if there is only right possible

} **else** **if**((cost[k][l]-cost[k][l-1]==1 || cost[k][l]-cost[k][l-1]==-1)

&& *stor*[k][l-1]!=0) {

*stor*[k][l]=cost[k][l]+*stor*[k][l-1];

}

}

}

}

//reverse track the path starting from our max row and col

**static** **int** makePath (**int** grid[][], **int** cost[][], **int** mmm, **int** nnn){

//find maximum element and index first

**int** max=-1;

**int** maxRow=-1;

**int** maxCol=-1;

**for** (**int** i=0; i<***NN***; i++) {

**for** (**int** j=0; j<***NN***; j++) {

**if** (grid[i][j]> max) {

maxRow=i;

maxCol=j;

max=grid[i][j];

}

}

}

*path*[maxRow][maxCol]=1;

//find the path by directions up or down

**while**(maxRow>=mmm && maxCol>=nnn) {

**if**(maxRow==mmm && maxCol==nnn)

**break**;

//move up if no left possible

**if**(maxRow==mmm) {

*path*[maxRow][maxCol-1]=1;

maxCol--;

}

//move left if no up possible

**else** **if**(maxCol==nnn) {

*path*[maxRow-1][maxCol]=1;

maxRow--;

}

//decide and move if two directions are possible

**else** {

//if both candidates are valid according to cost values

**if**((cost[maxRow][maxCol]-cost[maxRow][maxCol-1]==1 ||

cost[maxRow][maxCol]-cost[maxRow][maxCol-1]==-1)&&

(cost[maxRow][maxCol]-cost[maxRow-1][maxCol]==1 ||

cost[maxRow][maxCol]-cost[maxRow-1][maxCol]==-1 )) {

**if** (*stor*[maxRow][maxCol-1]>*stor*[maxRow-1][maxCol]){

*path*[maxRow][maxCol-1]=1;

maxCol--;

}

**else** **if**(*stor*[maxRow][maxCol-1]<*stor*[maxRow-1][maxCol]) {

*path*[maxRow-1][maxCol]=1;

maxRow--;

}

**else** {

/\* this part changes when we want to find all paths possible\*/

*path*[maxRow][maxCol-1]=1;

maxCol--;

}

}

// if only left is possible according to cost values, even though DP stor up is larger than left

**else** **if** ((cost[maxRow][maxCol]-cost[maxRow][maxCol-1]==1 ||

cost[maxRow][maxCol]-cost[maxRow][maxCol-1]==-1) &&

(*stor*[maxRow][maxCol-1])!=0){

*path*[maxRow][maxCol-1]=1;

maxCol--;

}

//if only up is possible according to cost values

**else** **if** ((cost[maxRow][maxCol]-cost[maxRow-1][maxCol]==1 ||

cost[maxRow][maxCol]-cost[maxRow-1][maxCol]==-1) &&

(*stor*[maxRow-1][maxCol])!=0){

*path*[maxRow-1][maxCol]=1;

maxRow--;

}

}

}

**return** max;

}

**public** **static** **void** main( String args[ ] ) {

**int** cost[ ][ ] = {

{ 6, 5, 2, 3, 1 },

{ 3, 4, 1, 4, 4 },

{ 1, 5, 6, 7, 8 },

{ 3, 4, 5, 8, 9 },

{ 3, 2, 2, 7, 6 }

};

//we have to do all elements as starting candidates

**for** (**int** i=0; i<***NN***; i++) {

**for** (**int** j=0; j<***NN***; j++) {

*minGrid*( cost, i, j );

//*printGrid*(*stor*, "DP");

**int** len=*makePath*(*stor*, cost, i, j);

//*printGrid*(*path*, "path: len "+len);

//if we found the largest length then save it

**if**(len>*maxLen*) {

*maxLen*=len;

*maxPath*=*path*;

}

//reset our stor and path again

*stor* = **new** **int** [***NN***][***NN***];

*path* = **new** **int** [***NN***][***NN***];

}

}

*printGrid*(*maxPath*, "longest path: len "+*maxLen*);

}

**static** **void** printGrid( **int** grid[ ][ ], String title ) {

System.***out***.println( "\n" + title );

**int** row;

System.***out***.print( "X |\t|" );

**for**( row = 0; row < ***NN***; row++ )

System.***out***.print( ( row + 1 ) + "\t| " );

System.***out***.print( "\n\n" );

**for**( row = 0; row < ***NN***; row++ ) {

System.***out***.print( ( row + 1 ) + " |\t|" );

**for**( **int** col = 0; col < ***NN***; col++ )

System.***out***.print( grid[ row ][ col ] + "\t| " );

System.***out***.println();

}

System.***out***.println();

} // printGrid

}

---------------------------------------------------------------------------------------------------------------------------

b) O(n^4)입니다. 본래 동적 프로그램으로 풀면 O(n^2)가 나오는데, 시작 점이 0,0이 아닐 수 있기 때문에 결국엔 (0,0)부터 (N-1,N-1)까지 각각 원소부터 골라서 그 속에 각자 O(n^2)의 함수를 풀어야 합니다. 물론 주어진 문제에서는 직관적으로 봐도 실제로 해봐도 0,0에서 시작하는 것이 빠른 것으로 나옵니다만, 만약 음의 가중치가 허용된다든가, 동 떨어져 있는 위치에 (3,3)이랑 (3,4)가 만약 값이 10000,10001이었으면 그 둘의 path가 가장 길기 때문에 각각 원소를 시작점으로 반복해서 돌리는 것이 필요하게 됩니다.

---------------------------------------------------------------------------------------------------------------------------

c) (제가 해보면서 시험지에 있는 예시로는 a랑 답이 1개로 똑같아서 모든 가능성의 path를 보여주기 위해서 cost 값을 임의로 바꿔놨습니다. 제가 확인하려는 것은 같은 시작점에서 시작해도 나중에 같은 최댓값으로 분기되는 경우랑, 동떨어져 있는 cost가 엄청 큰 경우 max값이 같은 경우, 그리고 다른 출발점에서도 최대 길이가 나오는 경우(이 경우 내에서도 똑같이 최댓값이 분기되는 것도 있음)을 확인하였습니다.)

**package** path;

**import** java.util.ArrayList;

/\* Maximum cost path solution using dynamic programming \*/

**public** **class** AllMaxCostPath {

**private** **static** **final** **int** ***NN*** = 5;

**private** **static** **int**[][] *stor* = **new** **int**[***NN***][***NN***];

**private** **static** ArrayList<**int**[][]> *paths* = **new** ArrayList<>();

**private** **static** **int** *maxLen* = -1;

**private** **static** ArrayList<**int**[][]> *maxPaths* = **new** ArrayList<>();

**static** **void** minGrid(**int** cost[][], **int** mmm, **int** nnn) {

// put cost in stor for our start

*stor*[mmm][nnn] = cost[mmm][nnn];

// fill in down DPs

**for** (**int** i = 1; i < ***NN***; i++) {

**if** ((cost[mmm][i] - cost[mmm][i - 1] == 1 || cost[mmm][i] - cost[mmm][i - 1] == -1)

&& *stor*[mmm][i - 1] != 0) {

*stor*[mmm][i] = cost[mmm][i] + *stor*[mmm][i - 1];

}

}

// fill in right DPs

**for** (**int** j = 1; j < ***NN***; j++) {

**if** ((cost[j][nnn] - cost[j - 1][nnn] == 1 || cost[j][nnn] - cost[j - 1][nnn] == -1)

&& *stor*[j - 1][nnn] != 0) {

*stor*[j][nnn] = cost[j][nnn] + *stor*[j - 1][nnn];

}

}

// fill in the rest of it

**for** (**int** k = 1; k < ***NN***; k++) {

**for** (**int** l = 1; l < ***NN***; l++) {

// according to cost values, if there are two candidates

**if** ((cost[k][l] - cost[k - 1][l] == 1 || cost[k][l] - cost[k - 1][l] == -1)

&& (cost[k][l] - cost[k][l - 1] == 1 || cost[k][l] - cost[k][l - 1] == -1)) {

// follow the larger stor value

**if** (*stor*[k - 1][l] >= *stor*[k][l - 1] && *stor*[k - 1][l] != 0) {

*stor*[k][l] = cost[k][l] + *stor*[k - 1][l];

} **else** **if** (*stor*[k - 1][l] < *stor*[k][l - 1] && *stor*[k][l - 1] != 0) {

*stor*[k][l] = cost[k][l] + *stor*[k][l - 1];

}

// according to cost values, if there is only down possible

} **else** **if** ((cost[k][l] - cost[k - 1][l] == 1 || cost[k][l] - cost[k - 1][l] == -1)

&& *stor*[k - 1][l] != 0) {

*stor*[k][l] = cost[k][l] + *stor*[k - 1][l];

// according to cost values, if there is only right possible

} **else** **if** ((cost[k][l] - cost[k][l - 1] == 1 || cost[k][l] - cost[k][l - 1] == -1)

&& *stor*[k][l - 1] != 0) {

*stor*[k][l] = cost[k][l] + *stor*[k][l - 1];

}

}

}

}

// reverse track the path starting from our max row and col

**static** **int** makePath(**int** grid[][], **int** cost[][], **int** mmm, **int** nnn) {

// find maximum element and index first

ArrayList<Integer> maxRows = **new** ArrayList<>();

ArrayList<Integer> maxCols = **new** ArrayList<>();

**int** multiPathItr = 0;

**int** max = -1;

**int** maxRow = -1;

**int** maxCol = -1;

**for** (**int** i = 0; i < ***NN***; i++) {

**for** (**int** j = 0; j < ***NN***; j++) {

**if** (grid[i][j] > max) {

maxRow = i;

maxCol = j;

max = grid[i][j];

}

}

}

maxCols.add(maxCol);

maxRows.add(maxRow);

**int**[][] path = **new** **int**[***NN***][***NN***];

path[maxRow][maxCol] = 1;

*paths*.add(path);

**while** (multiPathItr < *paths*.size()) {

path = *paths*.get(multiPathItr);

maxCol = maxCols.get(multiPathItr);

maxRow = maxRows.get(multiPathItr);

// find the path by directions up or down

**while** (maxRow >= mmm && maxCol >= nnn) {

**if** (maxRow == mmm && maxCol == nnn)

**break**;

// move up if no left possible

**if** (maxRow == mmm) {

path[maxRow][maxCol - 1] = 1;

maxCol--;

}

// move left if no up possible

**else** **if** (maxCol == nnn) {

path[maxRow - 1][maxCol] = 1;

maxRow--;

}

// decide and move if two directions are possible

**else** {

// if both candidates are valid according to cost values

**if** ((cost[maxRow][maxCol] - cost[maxRow][maxCol - 1] == 1

|| cost[maxRow][maxCol] - cost[maxRow][maxCol - 1] == -1)

&& (cost[maxRow][maxCol] - cost[maxRow - 1][maxCol] == 1

|| cost[maxRow][maxCol] - cost[maxRow - 1][maxCol] == -1)) {

**if** (*stor*[maxRow][maxCol - 1] > *stor*[maxRow - 1][maxCol]) {

path[maxRow][maxCol - 1] = 1;

maxCol--;

} **else** **if** (*stor*[maxRow][maxCol - 1] < *stor*[maxRow - 1][maxCol]) {

path[maxRow - 1][maxCol] = 1;

maxRow--;

} **else** {

/\*

\* stor[maxRow][maxCol - 1] == stor[maxRow - 1][maxCol] this part for find all

\* paths possible make bifurcation of two path

\*/

**int**[][] tmpPath = **new** **int**[***NN***][***NN***]; //make new path and copy original

*copy\_path*(path, tmpPath);

tmpPath[maxRow][maxCol - 1] = 1; // new path update

maxRows.add(maxRow);

maxCols.add(maxCol - 1);

*paths*.add(tmpPath);

path[maxRow - 1][maxCol] = 1; // original path update

maxRow--;

}

}

// if only left is possible according to cost values, even though DP stor up is

// larger than left

**else** **if** ((cost[maxRow][maxCol] - cost[maxRow][maxCol - 1] == 1

|| cost[maxRow][maxCol] - cost[maxRow][maxCol - 1] == -1)

&& (*stor*[maxRow][maxCol - 1]) != 0) {

path[maxRow][maxCol - 1] = 1;

maxCol--;

}

// if only up is possible according to cost values

**else** **if** ((cost[maxRow][maxCol] - cost[maxRow - 1][maxCol] == 1

|| cost[maxRow][maxCol] - cost[maxRow - 1][maxCol] == -1)

&& (*stor*[maxRow - 1][maxCol]) != 0) {

path[maxRow - 1][maxCol] = 1;

maxRow--;

}

}

}

multiPathItr++; // for next

}

**return** max;

}

**public** **static** **void** copy\_path(**int** source[][], **int** dest[][]) {

**for** (**int** i = 0; i < ***NN***; i++) {

**for** (**int** j = 0; j < ***NN***; j++)

dest[i][j] = source[i][j];

}

}

**public** **static** **void** main(String args[]) {

/\* new cost to check if this program is working well

we are willing to find paths that have same longest paths

in the same starting point, and also the the same longest

paths with different starting points\*/

**int** cost[][] = {{ 1, 3, 2, 3, 50 },

{ 3, 4, 1, 4, 4 },

{ 1, 5, 6, 7, 8 },

{ 3, 4, 5, 8, 9 },

{ 3, 2, 2, 9, 8 }};

// we have to do all elements as starting candidates

**for** (**int** i = 0; i < ***NN***; i++) {

**for** (**int** j = 0; j < ***NN***; j++) {

*minGrid*(cost, i, j);

// printGrid(stor, "DP");

**int** len = *makePath*(*stor*, cost, i, j);

// printGrid(paths, "path: len " + len);

// if we found the largest length then save it

**if** (len > *maxLen*) {

*maxLen* = len;

*maxPaths*.clear();

*maxPaths*.addAll(*paths*);

} **else** **if** (len == *maxLen*) {

*maxPaths*.addAll(*paths*);

}

// reset our stor and path again

*stor* = **new** **int**[***NN***][***NN***];

*paths* = **null**;

*paths* = **new** ArrayList<>();

}

}

*printGrid*(*maxPaths*, "longest path: len " + *maxLen*);

}

**static** **void** printGrid(**int** grid[][]) { // printGrid

**int** row;

System.***out***.print("X |\t|");

**for** (row = 0; row < ***NN***; row++)

System.***out***.print((row + 1) + "\t| ");

System.***out***.print("\n\n");

**for** (row = 0; row < ***NN***; row++) {

System.***out***.print((row + 1) + " |\t|");

**for** (**int** col = 0; col < ***NN***; col++)

System.***out***.print(grid[row][col] + "\t| ");

System.***out***.println();

}

System.***out***.println();

}

**static** **void** printGrid(**int** grid[][], String title) { // override printGrid for singlepath

System.***out***.println("\n" + title);

*printGrid*(grid);

System.***out***.println();

}

**static** **void** printGrid(ArrayList<**int**[][]> grids, String title) { // override printGrid for multipath

System.***out***.println("\n" + title);

**for** (**int** i = 0; i < grids.size(); i++)

*printGrid*(grids.get(i));

System.***out***.println();

}

}

2.--------------------------------------------------------------------------------------------------------------------

1)

**package** trav;

**import** java.util.ArrayList;

**public** **class** Tree {

//an arrayList to pick the middle of each section

**static** ArrayList<String> *preOrder*= **new** ArrayList<String>();

**static** **void** inToPre(String[] order, **int** left, **int** right) {

**if** (left>right)

**return**;

//recursively find the middle of middle left and middle right

**int** middle=(left+right)/2;

*preOrder*.add(order[middle]);

*inToPre*(order, left, middle-1);

*inToPre*(order, middle+1, right);

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

String[] inOrder={"d", "b", "e", "a", "f", "c", "g"};

**int** len= inOrder.length;

*inToPre*(inOrder, 0, len-1);

System.***out***.println(*preOrder*);

//just to check if the same works in larger tree

*preOrder*=**new** ArrayList<String>();

String[] newinOrder = {"h","d","i","b","j","e","k","a","l","f","m","c","n","g","o"};

len=newinOrder.length;

*inToPre*(newinOrder, 0, len-1);

System.***out***.println(*preOrder*);

}

}

---------------------------------------------------------------------------------

2) 중간에 middle 값을 list에 넣는 것은 상수 값이고, binary search 하듯이 시간 복잡성은 O(logN)입니다.

--------------------------------------------------------------------------------------------------------------------------

3) 1번의 해답에서 아이디어를 그대로 가져와서 완전 트리를 빈 값의 노드를 채워서 풀 트리로 만들어서 1번을 재사용 하는 방법을 썼습니다. 물론 그냥 1번의 풀트리를 넣어도 작동은 됩니다.

**package** trav;

**import** java.util.ArrayList;

**import** java.lang.Math;

**public** **class** Tree {

//an arrayList to pick the middle of each section

**static** ArrayList<String> *preOrder*= **new** ArrayList<String>();

**static** **void** inToPre(String[] order, **int** left, **int** right) {

**if** (left>right)

**return**;

//recursively find the middle of middle left and middle right

**int** middle=(left+right)/2;

//only append if order element is not empty

**if** (order[middle]!=**null**)

*preOrder*.add(order[middle]);

*inToPre*(order, left, middle-1);

*inToPre*(order, middle+1, right);

}

//a function to find the number of nodes needed for full tree

//k is 2\*\*n-1 if full tree, so finding the n is the point

**static** **int** findPowNum(**int** k) {

**for**(**int** i=0; i<=k; i++) {

**if**((**int**)Math.*pow*(2,i)<k+1 && k+1<=(**int**)Math.*pow*(2,i+1))

**return** (**int**)Math.*pow*(2,i+1);

}

**return** -1;

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

String[] inOrder={"d", "b", "e", "a", "f", "c"};

**int** len= inOrder.length;

//find the number of more spaces to fill to make full tree

**int** powNum = *findPowNum*(len);

String[] reOrder = **new** String[powNum-1];

//reOrder the array to full tree with null in empty spaces

**for** (**int** i =0; i<len; i++) {

reOrder[i]=inOrder[i];

}

len=reOrder.length;

*inToPre*(reOrder, 0, len-1);

System.***out***.println(*preOrder*);

//just to check if the same works in larger tree

*preOrder*=**new** ArrayList<String>();

String[] newinOrder = {"h","d","i","b","j","e","k","a","l","f","m","c","g"};

len=newinOrder.length;

powNum = *findPowNum*(len);

String[] reOrder2 = **new** String[powNum-1];

**for** (**int** i =0; i<len; i++) {

reOrder2[i]=newinOrder[i];

}

len=reOrder2.length;

*inToPre*(reOrder2, 0, len-1);

System.***out***.println(*preOrder*);

}

}

수업피드백:---------------------------------------------------------------------------------------------------------

1. 자바의 개념을 계속 중간에 훑어주시고 가주셔서 감사했습니다. 사실 수업으로는 자바를 처음 듣다 보니까 자바에 대해서 잘 모르는 상황이었는데, 첫 수업부터는 자바의 특성이나 중요한 개념들을 알짜배기로 요약해주신 점은 정말 감사합니다. 덕분에 이 수업을 가장 많이 복습하고 가장 많이 예습하려고 애썼습니다. 비록 중간고사는 제가 시험을 못 봤고, 집에서도 푸는 시험은 못 봤지만 그것이 반면교사가 되어 제일 많은 시간을 문제해결기법에 투자하게 되어 실력도 늘어난 것 같습니다. 아울러 교수님의 수업에서 예상치 못하게 개그 해주실 때가 재미있었습니다. 저는 재미있어서 가끔 웃었는데 저 많은 사람들 중에서 저 혼자만 웃는 거 같아서 그때부터 눈치보고 웃기 시작했습니다. 앞으로는 강의에 학생 수가 더 적으면 개그 해주셔도 편히 웃을 수 있을 것 같습니다. 그리고 알고리즘 수업이랑 자료구조 수업을 같이 듣고 있는데, 이 수업 덕분에 미리 예습하는 것 같아서 그 수업을 이해하는 데에도 많은 도움이 되었습니다. 코드도 직접 교수님이랑 같이 짜는 것이 가장 좋았습니다. 물론 코딩 할 때는 아예 도움 없이 스스로 낚싯대를 잡는 방법을 알려주는 것이 바람직하다고는 하지만 저처럼 전혀 갈피 못 잡는 학생들에게는 큰 도움이 되신 것 같아 정말 감사했습니다. 아울러 이번 수업에서 human sorting을 하면서 처음으로 컴퓨터 공학 수업에서 팀플 해본 경험이 가장 귀중하다고 생각합니다. 문과식 팀플에서는 서로서로 분담하고 서로서로 양보하고 서로서로 기분 안 나쁘게 얘기하려 하고 대강대강 하려는 경향이 있었지만, 이번 팀플에서는 팀원들과의 화합이 너무 맞았고, 서로 의견 대립도 많이 겪어가며 아주 열정적으로 했기 때문에 후회 없이 보람찬 팀플을 했던 것 같습니다. 이런 기회 주신 점 정말 감사하게 생각합니다.
2. 저희 조가 human sorting에서 결국 1등을 해냈지만, 그게 끝인 게 아쉬웠습니다. 그 방법을 설명해보고 싶었는데, 그렇지 못한 게 아직 아쉽습니다. Human Sorting 과제를 처음 내주실 적에 교수님께서는 컴퓨터라면 어떤 알고리즘들이 빠르겠지만, 인간이 직접 하면 다른 알고리즘이 필요할 수 있겠다고 하셔서 저희 조는 다른 조들도 할 법한 흔한 방법을 제일 안 하려고 했습니다. 다른 조는 예상대로, 평균 4명이라고 보면, 한 명 한 명이 백의 자리에 따라서, 즉 100~399, 400~599, 600~799, 800~999의 범위로 넘겨주며 배분하면 각자 그 속에서 또 MSD radix sort를 하는 개념과 비슷하게 insertion sort 나눈 후, 다시 서로에게 15장씩 배분하는 방법을 썼습니다. 저희 조도 이 방법을 처음에 생각해냈지만, 교수님께서 내주신 문제라면 너무 뻔한 문제는 냈을 리가 없다고 생각하고 단점들을 뽑아봤습니다. 우선 위의 방법은 눈으로 보기에는 편하지만 책상 위에 두고 할 수 없어 열 손가락으로 일일이 최대 10의 자리로 나누어가며 정렬을 해야 한다는 점, 만약 한자리의 백자리, 가령 500자리 숫자들이 몰려 있으면, 결국엔 한 명이 몰아서 하게 된다는 점(이 때문에 처음 나눠주기 전에 sorting하는 것에 따라서 결과가 달라지는 것을 종종 볼 수 있었습니다), 그리고 15장을 맞추고 조장이 스탑을 외치려면 다른 조원들이 15장 sorting이 완성되었는지 곁눈질 하거나(이건 명백히 규칙 위반인데 다른 조들이 하는 것을 몇 번 봤습니다.) 카드를 일부러 다 줬다가 다시 주는 방식으로 신호를 보내야한다는 점이 있었습니다. 그래서 이를 상쇄하려고 저희는 각자 15장을 알아서 sort 한 다음, 2명씩이 한 묶음이라고 보면 각 묶음의 한 명한테 30장을 몰아주고 30장을 가지고 mergesort를 했습니다. 그러면 두 명은 놀게 되는데, 이를 또 상쇄시키기 위해 두 명이 각자 30장을 섞는 도중에 600자리가 나오면 이미 sort해두고 있던 카드들을 노는 한 명한테 몰아주고, 계속 sorting 한 다음 나머지도 sort가 완성되면 그것을 또 다른 노는 한 명에게 몰아줍니다. 그럼 sorting 된 100~599의 카드들을 중 최소값의 15장은 첫 사람에게 주고 600~999의 카드 중에서 최대값을 가진 15장을 또 마지막 사람에게 주고 나머지 카드들은 2,3번째 사람들이 넘치는 사람이 넘치는 만큼 부족한 사람에게 서로 주면 자동으로 15,15,15,15장씩 가지게 되어 중간에 위치한 조장이 곁눈질 확인 없이 바로 확신하고 스탑 외칠 수 있게 했었습니다. 그 결과 저희는 처음에 배분 되었을 때 어느 백 자리에 몰리든 상관없도록 했고, 곁눈질해서 부정 행위하는 것 없이 스탑을 외칠 수 있었고, 직접 10진수를 10 손가락으로 분류하는 것보다는 화투 패 나누듯이 큰 거 작은 거만 비교하는 방법으로 두 손을 이용해서 속도를 올릴 수 있었습니다. 그런데 마지막에 마무리하실 때는 mergesort하면 놀게 되니까 각자 쉬지 않는 선에서 빠르게 sorting하는 것이 답이었다고 하시고 수업을 엄청 일찍 끝내셔서 한편으로는 너무 아쉬웠습니다. 다른 조들은 결국 원래 그 방법으로 했고, 그들의 sorting 시간을 결정한 것은 그들의 알고리즘이 아니라 각자의 손 기술과 처음 sorting 되었을 때 골고루 분배된 정도에 있었기 때문입니다. 다음에는 저희 조처럼 독특하게 하면서도 빠르게 한 조가 있으면 발표할 기회가 있었으면 좋겠습니다. 저희를 위해서 일일이 코딩하시고 피드백도 잘 받아주시고, 저희 재미있게 가르쳐주시려고 하셔서 고생 많으셨습니다! 성적과는 무관하게 보람찬 수업이었습니다. 감사합니다.