Midterm Summary

- 201711237 유재덕

recursion

1. terminate condition
2. reduce the size of problem

- example
    1. hanoi tower(recursive -> non-recursive)

```
void hanoi(int block_size, int from, int by, int to) {
    if (n == 1)
            move(from, to);
            // terminate condition
    else {
            hanoi(n - 1, from, to, by);
            // size : n -> n-1
            move(from, by);
            hanoi(n - 1, by, from, to);
    }
}
```

Graph(model)

1. G = vertex + edge
2. edge : connection between vertex
3. Representation
    1. Adjacency Matrix

```
map             Graph
                 |A B C D E F G
                 ―――――――――
A - B - E       A|1 1 1 0 0 0 0
|               B|1 1 0 0 1 0 0
C - D           C|1 0 1 1 0 1 0
|   |           D|0 0 1 1 0 0 1
F   G           E|0 1 0 0 1 0 0
                F|0 0 1 0 0 1 0
                G|0 0 0 1 0 0 1
```

```
void input_adjmatrix(int **map, int *V, int *E) {
    int i, j, k;
    char line[3];
```

```
    // 마지막 칸에는 NULL이 들어간다.
    printf("num of vertex : ");
    scanf_s("%d", V);
    printf("num of Edges : ");
    scanf_s("%d", E);
    for (i = 0; i < *V; i++)
        map[i][i] = 1;
        // 매트릭스 내에서 서로 같은 노드끼리는 무조건 연결되어있다고 가정.
    for (k = 0; k < *E; k++) {
        printf("\nInput two node consist of edge : ");
        scanf("%s", line);
        // input을 받음
        i = name2int(line[0])-1;
        j = name2int(line[1])-1;
        map[i][j] = 1;
        map[j][i] = 1;
        // 입력받은 위치의 매트릭스 값을 1로 변환
    }

}
```

2. Adjacency List

```
map             Graph
A - B - E       A - C - B
|               B - E - A
C - D           C - F - D - A
|   |           D - G - C
F   G           E - B
                F - C
                G - D
```

```
void input_adjlist(node *a[], int *V, int *E) {
    char vertex[3];
    int i, j;
    node *t;
    printf("\nInput number of node & edge\n");
    scanf("%d %d", V, E);
    // vertex의 개수와 edge의 개수를 입력받는다.
    for (j = 0; j < *E; j++) {
        printf("\nInput two node consist of edge →");
        scanf("%s", vertex);
        i = name2int(vertex[0]);
        t = (node *)malloc(sizeof(node));
        t->vertex = name2int(vertex[1]);
        t->next = a[i];
        // i번째 vertex의 graph 앞에 만든 node를 붙인 후
        a[i] = t;
        // i번째 vertex에 관한 graph에 이어 붙인 후 head를 옮김
```

```
        i = name2int(vertex[1]);
        t = (node *)malloc(sizeof(node));
        t->vertex = name2int(vertex[0]);
        t->next = a[i];
        a[i] = t;
        // j번째 vertex에 관한 graph에 이어 붙인 후 head를 옮김
    }
}
```

Graph Search

1. DFS : 스택 활용
    - recursive : 알아서 짜든가 공부하든가 하자.
    - non-recursive

```
void nrDFS_adjlist(node *a[], int V)
{
    node *t;
    int i;
    init_stack();
    for (i = 0; i < V; i++)
        check[i] = 0;
        // check 초기화. check는 해당 vertex를 stack에 push할때 set된다.
    for (i = 0; i < V; i++)
    {
        if (check[i] == 0)
        // 만약 i vertex가 한번도 push되지 않았다면
        {
            push(i);
            // i(root)를 집어넣어줌
            check[i] = 1;
            // i가 push되었으므로 check set
            while (!is_stack_empty())
            // 스텍이 빌때까지
            {
                i = pop();
                visit(i);
                // pop 후 방문
                for (t = a[i]; t != NULL; t = t->next)
                        if (check[t->vertex] == 0)
                        // 만약 다음 graph에 있는 vertex가 한번도 push 안
된으면
                        {
                            push(t->vertex);
                            check[t->vertex] = 1;
                            // push하고 check를 set
                        }
            }
        }
    }
}
```

2. BFS : 큐 활용
    - recursive : 알아서 하기
    - non-recursive

```
void BFS_adjmatrix(int *a[], int V) {
    int i, j, k;
    node *t;
    init_queue();
    for (i = 0; i < V; i++) check[i];
    // 모든 vertex를 아직 put되지 않았다고 설정해준다.
    for (i = 0; i < V; i++) {
        if (check[i] == 0){
        // i(root)가 한번도 put되지 않았다면
            put(i);
            // 큐에 집어넣음
            check[i] = 1;
            // 큐에 집어넣었으므로 check를 set해준다.
            }
        while (!queue_empty()) {
        // 큐가 빌 때까지
            k = get(i);
            // 데이터 빼오고
            t = a[k];
            // t를 통해 빼온 데이터에 관한 그래프를 접속
            visit(k);
            while (t) {
                // t가 존재할때까지(NULL이 아닐떄까지)
                if (check[t->vertex] != 1) {
                // 만약 연결된 vertex가 한번도 put되지 않았다면
                    put(t->vertex);
                    check[t->vertex] = 1;
                    // 집어넣고 check를 set
                }
                t = t->next;
                // 다음 데이터로 넘어감. 아마 수업에선 for loop로 구현했던
걸로 기억함
            }
        }
    }
}
```

Biconnectivity

- Articulation Point : 해당 vertex를 제외할 때 서로 다른 두 개의 그래프로 분리되는 지점.
- 수업 내에선 DFS 기반의 탐색으로 구현
    1. 탐색 순서를 활용한다.
    2. min : 자신의 탐색 순서(check[vertex]) -> 자신과 자식 노드들 중 최소 탐색 순서
    3. m : 자식 노드들 중 최소 탐색 순서

4. 만약 check[vertex]가 min값보다 작거나 같으면 해당 vertex는 AP이다.

- root의 자식의 개수?
  - 한 개 : root가 있으나 없으나 상관 없다
  - 두 개 이상 : root가 곧 단결점이 된다.
  - root는 탐색 순서가 항상 더 작으니 위의 알고리즘을 적용할 수 없다. 따라서 자식 개수를 통해 단결점인지 파악한다.

```
int AP_recur(node *a[], int i) {
    node *t;
    int min, m;

    check[i] = min = ++order;
    // 1. min <= 탐색 순서, 2. check[i] <= min
    for (t = a[i]; t != tail; t = t->next) {
    // a[i]는 input으로 받은 i vertex에 관한 그래프를 의미한다.
        if (i == 0 && check[t->vertex] == 0)
            son_of_root++;
            // root가 단결점인지 아닌지 판단할때 쓴다.
        if (check[t->vertex] == 0) {
        // 기준 vertex와 tree edge로 연결되어 있을 때 해당 알고리즘 수행
        // 만약 해당 vertex가 put되지 않았다면
            m = AP_recur(a, t->vertex);
            // 자식 node들 중 가장 낮은 탐색 순서를 반환받는다.
            if (m < min) min = m;
            // 만약 반환받은 값과 현재 min(자신의 탐색순서) 값 중 더 작은
값을 min에 대입
            if(m >= check[i] && i != 0)
                printf("* %c % 2d : %d\n", int2name(i),check[i], m);
                // 만약 자식 node들의 탐색순서 중 최소값이 자신의 탐색순서
보다 크다면 해당 node는 단결점이다.
            else
                printf("  %c % 2d : %d\n", int2name(i),check[i], m);
                // 그 외엔 단결점 아님
        }
        else
        // non-tree edge일때 해당 알고리즘 수행
        // 만약 이전에 put된적 있는 vertex라면
            if (check[t->vertex] < min)
                min = check[t->vertex];
                // 해당 node의 참조 순서와 본인과 tree edge로 연결된 자식
들의 탐색 순서 중 가장 작은 값 중 더 작은 값을 가져간다.
    }
    return min;
    // tree edge와 non-tree edge 포함해 연결되어있는 vertex 중 가장 낮은
탐색 순서를 반환한다.
}
```

heap

- 특정 값을 기준으로 우선순위를 결정해주는 자료형.

PFS

- weight를 기준으로 heap에 vertex를 정렬한다.
- 따라서 모든 vertex마다 tree 내에서 부모 node가 어떤 vertex인지 저장하는 배열이 필요함.
- visit & unvisit : 양수와 음수로 visit과 unvisit을 결정. heap 알고리즘 내의 모든 vertex들은 unvisit이기 때문에 계산을 반대로 해야한다.
- parent : 부모가 중간에 바꼈을때 알고리즘을 잘 보자.

```
#define _CRT_SECURE_NO_WARNINGS

#include<stdio.h>
#include<stdlib.h>
#define UNSEEN -INT_MAX
#define MAX_NODE 100
#define name2int(c) (c-'A')
#define int2name(c) (c+'A')
#define tail NULL

typedef struct _node {
    int vertex; // 연결되는 노드
    int weight; // 가중치
    struct _node *next;
}node;

typedef struct _path {
    int vertex;
    struct _path *next;
}path;

node graph[MAX_NODE];   // 그래프
path *p;
int check[MAX_NODE];   // 가중치 저장
int h[MAX_NODE];   // 힙
int parent[MAX_NODE];   // 부모 노드 저장
int nheap = 0;
FILE *fp;   // 파일 읽어올때 사용함

void input_adjlist(node *a[], int *V, int *E);
// 그래프 설정 함수. 해당 코드에선 graph.txt를 받아 쓴다.
// a : graph, V : vertex size, E : edge size
void print_list(node *a[], int V);
// 프린트 출력 함수. 단 모든 값을 NULL까지 밀어버리므로 head를 복사해서 써야됨
// a : graph, V : vertex size
void pq_init();
// heap 초기화 함수
int pq_empty();
// heap 내의 데이터를 비우는 함수
void upheap(int *a, int k);
// up heap : 주어진 index를 기준으로 위로 올라가며 정렬
// a : heap, k : index num
void downheap(int *a, int k);
// down heap : 주아전 index를 기준으로 아래로 내려가며 정렬
```

```
// a : heap, k : index num
void insert(int *a, int *N, int v);
// heap에 데이터를 추가함
// a : heap, N : index size, v : weight
void adjust_heap(int h[], int n);
// 모든 index에 대해 down heap을 실행
// h : heap, n : index size
int pq_extract(int *a);
// root의 값을 빼오는 함수
// a : heap
int pq_update(int h[], int v, int p);
// 1. 새로운 edge의 가중치가 기존 edge의 가중치보다 적을 경우 해당 node에 대한
queue의 edge를 새로운 edge로 바꿈
// 2. 만약 기존에 없던 node에 대한 edge가 있을경우, 해당 edge를 heap에 집어넣는
다.
// h : heap, v : vertex num, p : weight
void PFS_adjlist(node *g[], int V);
// PFS알고리즘
// g : graph, V : vertex size
void visit(int i);
void print_heap(int h[]);
void print_cost(int weight[], int index);

void init_path(int v);
void push_path(int srcv, int curv);
void clear_path(int v);
int main() {
    int V, E;

    printf("\nOriginal Graph\n");
    input_adjlist(&graph, &V, &E);
    // get graph
    print_list(&graph, V);

    init_path(V);

    printf("\nVisit order of Minimum Spanning Tree\n");
    PFS_adjlist(graph, V);
    // algorithm
    for (int i = 1; i < V; i++) {
        printf("\nparent of %c : %c", (char)i + 65, (char)parent[i] + 65);
        // print parent[]
        print_cost(parent, i);
        // get minimum cost of node i
        printf("\n");
    }
    while (1);
}

void input_adjlist(node *g[], int *V, int *E)
{
    char vertex[3];
    int i, j, w;
    node *t;
```

```
    fp = fopen("graph.txt", "r");

    fscanf(fp, "%d %d", V, E);
    for (i = 0; i < *V; i++)
        g[i] = NULL;
    for (j = 0; j < *E; j++) {
        fscanf(fp, "%s %d", vertex, &w);
        i = name2int(vertex[0]);
        t = (node *)malloc(sizeof(node));
        t->vertex = name2int(vertex[1]);
        t->weight = w;
        t->next = g[i];
        g[i] = t;
        i = name2int(vertex[1]);
        t = (node *)malloc(sizeof(node));
        t->vertex = name2int(vertex[0]);
        t->weight = w;
        t->next = g[i];
        g[i] = t;
    }
    fclose(fp);
}

void print_list(node *a[], int V) {
    // this function push all pointers of 'a' until NULL, so make same data
    int i = 0;
    node *x = (node*)malloc(sizeof(node));
    while (i < V) {
        x = a[i];
        if (x) {
            printf("%c\t", i + 65);
            while (x) {
                printf("%2c", x->vertex + 65);
                x = x->next;
            }
            printf("\n");
            i++;
        }
    }
    free(x);
}

void print_heap(int h[]) {
    int i;
    printf("\n");
    for (i = 1; i <= nheap; i++)
        printf("%c:%d ", int2name(h[i]), check[h[i]]);
    // print heap
}


void pq_init() {
    nheap = 0;
```

```
    // set node size of heap zero
}

int pq_empty() {
    if (nheap == 0) return 1;
    return 0;
    // if empty : true
    // else : false
}

void adjust_heap(int h[], int n) {
    int k;
    for (k = n / 2; k >= 1; k--)
        downheap(h, k);
    // downheap all nodes except leaf nodes
}

void upheap(int *a, int k) {
    int v;
    v = a[k];
    while (check[h[k / 2]] < check[v] && k / 2>0) {
        // if weight of children of k is smaller than weight of v,
        // ps. weight of unvisited nodes are minus
        a[k] = a[k / 2];
        k = k >> 1;
    }
    a[k] = v;
}

void downheap(int *a, int k) {
    // N : node number of tree
    int i, v;
    v = a[k];
    while (k <= nheap / 2) {
        i = k << 1;
        if (i < nheap&& check[a[i]] < check[a[i + 1]]) i++;
        if (check[v] > check[a[i]]) break;
        a[k] = a[i];
        k = i;

    }
    a[k] = v;
}

int pq_extract(int *a) {
    // get root node and down heap
    int v = a[1];
    a[1] = a[nheap--];
    downheap(a, 1);

    return v;
}

void insert(int *a, int *N, int v)
```

```
{
    a[++(*N)] = v;
    upheap(a, *N);
}

int pq_update(int h[], int v, int p)
{
    if (check[v] == UNSEEN) {
        // if node is unseend before
        h[++nheap] = v;
        check[v] = p; // store the weight
        upheap(h, nheap);
        return 1;
        // add new heap
    }
    else {
        if (check[v] < p) {
            // node is seened before and edge has smaller wieght
            // ps. weights are stored in negative form
            check[v] = p;
            adjust_heap(h, nheap);
            // change edge of node
            return 1;

        }
        else
            return 0;
    }
}

void visit(int i) {
    printf("\nvisit %c\n", i + 65);
}

void PFS_adjlist(node *g[], int V)
{
    int i;
    node *t;
    pq_init();
    for (i = 0; i < V; i++) {
        check[i] = UNSEEN; // set nodes as unseen
        parent[i] = 0; // initialize a tree
    }
    // init check & parent
    for (i = 0; i < V; i++) {
        if (check[i] == UNSEEN)
            parent[i] = -1; // set the root
        pq_update(h, i, UNSEEN);
        // push i
        while (!pq_empty()) {
            print_heap(h);
            // print current heap
            i = pq_extract(h);
            // pop root(heap[0]) of heap
```

```
            check[i] = -check[i];
            // if check > 0 -> visited node
            // else         -> unvisited node
            visit(i);
            // print current node
            for (t = g[i]; t != NULL; t = t->next)
                // while graph of current node is end
                if (check[t->vertex] < 0)
                    // if node of graph is unvisited
                    if (pq_update(h, t->vertex, -t->weight))
                        // true : node of graph is seened before
                        // false : node of graph is unseened before
                        parent[t->vertex] = i;   // change parent node
        }
    }
}

void print_cost(int parent[], int index) {
    int i = index, result = 0;
    // needs to print index after, so make same data
    path *t = p + index;

    while (i > 0) {
        // while i > 0
        result += check[i];
        // add check(weight) of i at result
        push_path(index, i);
        i = parent[i];
        // change parent(i)
    }
    printf("\n%c까지 가는 가중치의 총합 : %d", (char)index + 65, result);
    printf("\npath : ");
    while (t) {
        printf("  %c", t->vertex+'A');
        t = t->next;
    }
}
void init_path(int v) {
    int i;
    p = (path*)malloc(sizeof(path)*(v+1));
    for (i = 1; i <= v; i++) {
        p[i].next = NULL;
        p[i].vertex = 0;
    }
}

void push_path(int srcv, int curv) {
    path *t = (path*)malloc(sizeof(path));
    t->vertex = curv;
    t->next = p[srcv].next;
    p[srcv].next = t;
}
```

Kruskal

- weight를 기준으로 heap에 edge를 정렬한다.
- cycle을 만들면 tree를 만들지 못한다. 따라서 사이클을 만들면 안됨.
- union : 만약 두 대상 vertex가 같은 tree 내에 존재한다면 해당 vertex들을 연결하면 cycle이 형성된다.
    - 해당 알고리즘을 두 vertex가 속한 tree 내의 root node를 사용해 수행하므로 부모 node를 저장하는 알고리즘이 필요하다.

```
#define _CRT_SECURE_NO_WARNINGS

#include<stdio.h>
#include<stdlib.h>
#define UNSEEN -INT_MAX
#define MAX_NODE 50
#define MAX_EDGE 100
#define ONLY_FIND 0
#define UNION 1
#define tail NULL
#define name2int(c) (c - 'A')
#define int2name(c) (c + 'A')

typedef struct edge {
    int v1, v2;
    int weight;
}edge;

int parent[MAX_NODE]; // tree structure for representing the set
int height[MAX_NODE]; // store the height of the tree -> useless in
this code
int cost = 0;
int nheap = 0; // # of elements in the heap
int h[MAX_NODE];
edge *Edge;
FILE *fp;
void visit(int e);
void input_edge(int *V, int *E);
// get edge imformation in graph.txt
void kruskal(int V, int E);
// kruskal algorithm
void pq_insert(int v);
// insert leaf edge
int pq_extract();
// extract root of heap
void find_init(int vertex_size) {
    int i;
    for (i = 0; i < vertex_size; i++)
        parent[i] = -1;
}
// init find
void pq_init() { nheap = 0; }
// init heap
void visit(int e);
```

```
// visit algorithm
int pq_empty() {
    if (nheap == 0) return 1;
    return 0;
}
// clear data in heap
void upheap(int k);
// upheap
void downheap(int k);
// downheap
int find_set(int elem, int asso, int flag);
// find root node and compare
void union_set(int elem, int asso);
// union src and dst

void main()
{
    int V, E;
    fp = fopen("graph.txt", "rt");
    input_edge(&V, &E);
    printf("\n\nVisited edge of minimum spanning tree\n");
    kruskal(V, E);
    printf("\n\nMinimum cost is \n %d\n", cost);
    fclose(fp);
    while (1);
}

void input_edge(int *V, int *E) {
    char vertex[3];
    int i, w;
    printf("\nInput number of nodes and edges\n");
    fscanf(fp, "%d %d", V, E);
    Edge = (edge*)malloc(sizeof(edge)*(*E));
    for (i = 0; i < *E; i++) {
        printf("\nInput two nodes consisting of edge and its weight
→");
        fscanf(fp, "%s %d", vertex, &w);
        vertex[2] = NULL;
        Edge[i].v1 = name2int(vertex[0]);
        Edge[i].v2 = name2int(vertex[1]);
        Edge[i].weight = w;
    }
}

int find_set(int elem, int asso, int flag) {
    int i = elem, j = asso;
    while (parent[i] >= 0)
        i = parent[i];
        // find root node of i
    while (parent[j] >= 0)
        j = parent[j];
        // find root node of j
    if ((flag == UNION) && (i != j))
        union_set(i, j);
```

```
        // if different and union mode, then union
        // WARNING! BOTH OF I AND J IS ROOT
    return (i != j);
}

void kruskal(int V, int E) {
    int n, val = 0;
    find_init(V);
    pq_init();
    for (n = 0; n < E; n++)
        pq_insert(n);
        // before kruskal algorithm, put edges 0 to E
    n = 0;
    // change useage : count -> number of edge
    while (!pq_empty()) {
        // until heap empty
        val = pq_extract();
        // get root of heap
        if (find_set(Edge[val].v1, Edge[val].v2, UNION))
        // get v1 and v2 by edge number, then find_set by union mode
        {
            visit(val);
            n++;
            // if union, then new tree edge connected
        }
        if (n == V - 1)
            break;
            // if n is V-1, then all nodes are connected
    }
}

void union_set(int elem, int asso) {
    parent[elem] = asso;
    //
    printf("\tcombine!!\t");
}

void pq_insert(int v) {
    h[++(nheap)] = v;
    upheap(nheap);
}

int pq_extract() {
    // get root node and down heap
    int v = h[1];
    h[1] = h[nheap--];
    downheap(1);

    return v;
}

void upheap(int k) {
    int v;
    v = h[k];
```

```
    while (Edge[h[k / 2]].weight >= Edge[v].weight && k / 2>0) {
        // if weight of children of k is smaller than weight of v,
        // ps. weight of unvisited nodes are minus
        h[k] = h[k / 2];
        k = k >> 1;
    }
    h[k] = v;
}

void downheap(int k) {
    // N : node number of tree
    int i, v;
    v = h[k];
    while (k <= nheap / 2) {
        i = k << 1;
        if (i < nheap&& Edge[h[i]].weight > Edge[h[i + 1]].weight) i++;
        if (Edge[v].weight <= Edge[h[i]].weight) break;
        h[k] = h[i];
        k = i;
    }
    h[k] = v;
}

void visit(int e) {
    printf("%c%c ", int2name(Edge[e].v1),
        int2name(Edge[e].v2));
    cost += Edge[e].weight;
}
`
```