2014 Spring

Problem Solving Final Exam

1	2	3	4	5	6	7	Total

학번	
이름	

1. 다음 상황들에 대해 주어진 자료구조 중 어떤 자료구조가 가장 적합한지 고르고, 이유를 설명하시오.

1) 수강신청을 위한 웹페이지를 제작할 때, 특정 과목에 대해서 신청자가 몰릴 시 선착순으로 신청되는 것을 보장하기 위한 솔루션을 제작해야할 경우.
2) 쇼핑몰에서 종류가 고정된 상품의 수시로 변화하는 재고 수를 관리하기 위한 프로그램을 작성해야 할 경우.
3) 웹 브라우저의 기능 중 “뒤로가기” 기능을 구현해야 할 경우.
4) 하루에 수천명이 가입하고 탈퇴하는 인터넷 포탈의 회원 관리 프로그램을 작성해야할 경우.

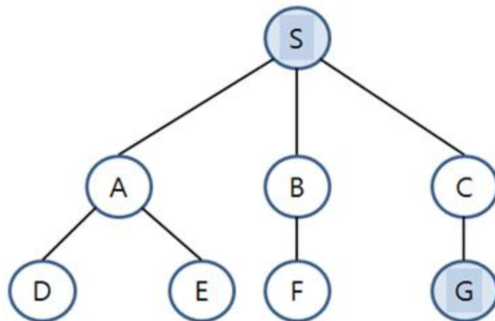
Data structures: FIFO-Queue, Stack, Array, Linked list,

1) FIFO Queue 선착순을 보장해주기 위해서는 FIFO Queue 가 적합한 자료구조이다.
2) 종류가 고정되었고, 그 값이 수시로 변할 경우, 빠른 참조를 위해서 Array 가 적합하다.
3) 뒤로 가기 기능은 바로 직전에 있는 페이지로 이동하는 것이다. Stack 에 push 하여 pop 하는 시맨틱을 그대로 따르므로 스택이 적합하다.
4) 입출력이 빈번한 데이터의 경우, linked list 가 적합하다.

2. 그래프를 순회하기 위한 대표적인 알고리즘에는 DFS 와 BFS 가 있다. 이중 BFS 는 모든 인접 노드를 탐색한 뒤, 다음 단계의 인접 노드를 탐색하는 방법이다. 그래프를 순회하기 위해서 사용되는 자료 구조에는 stack 과 queue 두 종류가 있다. 각 자료구조가 제공하는 동작은 다음과 같다.

Stack : push(s,v) : 스택 S 에 v 를 삽입한다. pop(s) : 스택 S 로부터 가장 나중에 삽입된 값을 가져온다.
queue : enqueue(q,v) : 큐 Q 에 v 를 삽입한다. dequeue(q) : 큐 Q 로부터 가장 먼저 삽입된 값을 가져온다.

Stack 혹은 Queue 를 사용하여 아래의 그래프를 BFS 방식으로 탐색하려고 한다.(S→G)



1) 위의 자료구조 중에 하나를 선택하고, BFS 과정 중에 stack 이나 queue 내에 존재하는 node 를 단계별로 표시하시오.

BFS **Queue (S A)** **Queue (S A B)** **Queue (S A B C)** **Queue (A B C)** **Queue (A B C D)** **Queue (A B C D E)** **Queue (B C D E)** **Queue (C D E)** **Queue (C D E F)** **Queue (D E F)** **Queue (D E F G)** **Queue (E F G)**	ex) DFS start : stack or queue (S) step1 : stack or queue (S A) step2 : stack or queue (S A D) …

Queue (F G) **End : stack or queue (G)**	

2) 다음 BFS를 위한 코드의 빈칸을 채우시오. (단 위에서 선택한 자료구조의 함수를 사용하여야 한다.)

```
bool discovered[MAXV]; //Array of discovered vertex
void bfs(graph *g, int start)
{
        stack or queue a;
        int v;
        int i;
        (1);
        discovered[start]=TRUE;
        while (empty(&a) == false)
        {
                (2);
              for (i=0; i<g->degree[v];i++)
                  if(valid_edge (g->edge[v][i]) == TRUE){
                      if(discovered[g->edges[v][i]] == FALSE){
                                  (3);
                              discovered[g->edges[v][i]] = TRUE;
                      }
        }
}
```

Hint : g->edges[v][]에는 v 와 연결된 모든 정점이 존재한다.
g->degree[v]에는 v 에 연결된 정점의 수가 존재한다.

답 :

(1) : enqueue(a,v)

(2) : v = dequeue(a)

(3) : enqueue(a,g->edge[v][i])

3. 높은 고층 빌딩에 엘리베이터 한대가 있는 경우를 가정해 보자. 이러한 엘리베이터를 이용할 때, 탑승객들이 서로 가기 원하는 층 버튼을 연속으로 누르면 비효율 적일 것이다. 따라서 엘리베이터는 탑승객들이 내리길 원하는 층을 분석해서 n 개의 층에서만 서게 된다. 이렇게 정지할 층을 계산하기 위하여, 우리는 걸어서 올라가거나 내려가야 하는 사람 수를 최소화할 층에 정지하여야 할 것이다.

1) i 층까지, j 번 멈추면서 모든 승객을 운반하기 위한 최소 비용을 $\mathrm{m}_{\mathrm{i,j}}$ 으로 정의한다. 그렇다면, (j-1) 번째 멈췄던 위치는 i 층보다 낮으며, 이를 k 층이라고 가정하자. 그렇다면, i 층에 내릴 사람들은 k 층 위로 올라가는 사람들에 대해서만 고려하면 된다. 이러한 사실을 바탕으로 이 문제를 해결하기 위해 재귀적 알고리즘을 사용할 수 있다. $walk(a,b)$함수는 a 층에서 사람들이 멈춘 후, b 층에서 멈출 때 승객들이 걸어서 이동하는 층수의 총합을 나타내는 함수이다. walk 함수를 이용하여 아래 점화식의 (1) , (2), (3) 을 채우시오.

$$\mathrm{m}_{\mathrm{i,j}} = \min_{\{k=0\ldots i-1\}} \{m_{k,j-1} - (1) + (2) + (3)\}$$

답 :

(1) : walked(k,∞)

(2) : walked(k,i)

(3) : walked(I,∞)

(2)와 (3)은 순서가 바뀌어도 무방하다.

2) $walked(a, b)$함수를 완성하시오. (최대 10 줄을 넘지 않아야 한다.)

```
int walked(int previous, int current)
{
    int nsteps=0;                  // 승객들이 걸어서 이동한 층수
    int i;
    int people=50;                 // 탑승객수
    int stops[50];                 //  승객이 내리고자 하는 층이 입력되어 있는 배열

for (i=0; i<nriders;i++)
  if ((stops[i] > previous) && (stops[i] <=current))
     nsteps += min(stops[i]-previous, current-stops[i]) // if 문으로 작성해도 무방.









    return(nsteps);
}
```

4. 사전에 있는 두 단어에 대해서, 한 단어 x 로부터 다른 단어 y 로의 변형이 하나의 문자를 수정하거나 삭제하거나 덧붙임으로써 완성된다면, 두 단어 x 와 y 는 edit step 으로 한 edit step 안에 있다고 할 수 있다.

Ex) sing -> song 은 한 edit step 에 있다.

그리고 한번의 edit step 을 통해 표현할 수 있는 단어들을 순서대로 나열한 것을 edit step ladder 라고 한다.

Ex) king, sing, song, // an edit step ladder

아래와 같은 단어 집합이 있을 때, 다음 물음에 답하시오.

Word set: dig dog fig fin fine fog wind wine

1) 최대 길이를 갖는 edit step ladder 를 쓰시오.

Dig-fig-fin-fine-wine

2) edit step ladder 를 발견하는 과정을 스택과 그래프 표현을 사용하여 설명시오. (단, 그래프의 정점은 원, 간선은 실선으로 표현하며, 각 과정을 순서대로 표현하시오.)

DFS 방법으로 문제를 표현하면 된다.

1) 새 단어가 들어올 때마다 그래프 전체를 순회하고,
2) 이전 단어로 돌아갈 때 stack 에서 pop 하며
3) 마지막 wine 을 찾을 때 fine 과 wind 에서 둘 다 edge 가 연결됨

채점기준

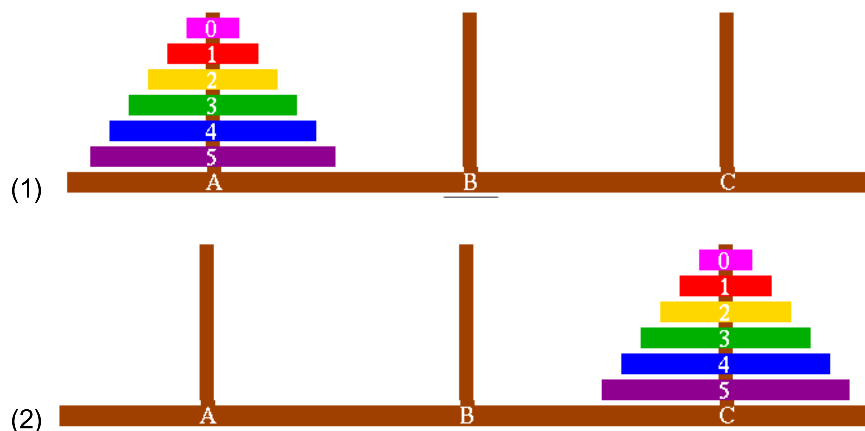
전체를 다 표현할 필요 없이, 과정의 일부만 보이되, 위의 세 가지 내용이 다 있을 경우 정답

각각에 대해 부분점수 줄 것.

5. 하노이의 탑은 재귀적인 방법을 통해 풀 수 있는 문제 중 가장 유명한 문제이다. 아래의 그림과 같이 세 개의 기둥과 이 기둥에 꽂을 수 있는 크기가 다양한 원판들이 있고, 한 기둥에 원반들이 작은 것부터 큰 순서 대로 쌓여 있다. 문제의 목적은 다음 두 가지 조건을 만족시키면서, 한 기둥에 꽂힌 원반들을 그 순서 그대로 다른 기둥으로 옮겨서 다시 쌓는 것이다.

1) 한번에 하나의 원반만 옮길 수 있다.
2) 큰 원반이 작은 원반위에 있을 수 없다.

다음 주어진 그림과 코드를 보고 문제에 대한 답안을 작성하시오.




```
void move_tower(disk, src, dst, spare)
If(disk == 0)
   move(disk, src, dst);
else {
   move_tower(   (1)   );
   move(disk, src, dst);   //src 로부터 dst 로 원반을 옮긴다.
   move_tower(   (2)   );
}
```

disk : 디스크의 갯수,

src : 해당 기둥으로부터 (그림상의 A 기둥)

dst: 해당 기둥으로 원반들을 이동 (그림상의 C 기둥)

spare : 그림상의 B 기둥

1) 그림 (1)에서 그림 (2)까지 총 몇번의 원반 이동이 일어났겠는가? 그 최소값을 구하시오.

63 번

2) 코드의 재귀 호출에 들어갈 알맞은 인자를 각각 고르시오.

(a) disk–1, src, spare, dst
(b) disk-1, dst, src, spare
(c) disk-1, spare, dst, src
(d) disk-1, spare, src, dst

(1)a

(2)d

3) 이러한 재귀 함수는 어째서 답을 발견할 수 있는가? 그 이유를 서술하시오.

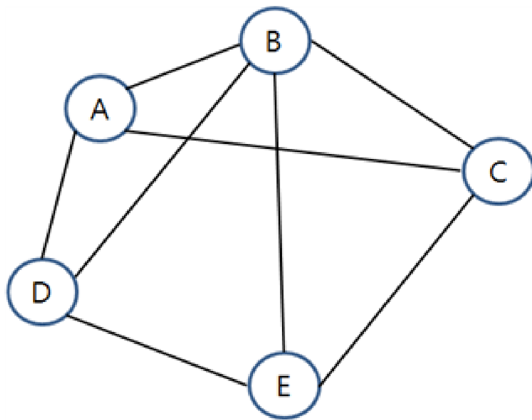
귀납적인 증명방법으로 문제의 답을 발견할 수 있음을 보일 수 있다.
먼저 원반이 1 개뿐인 문제를 가정하자. 해당 원반을 기둥 C 로 한번 옮기는 것으로 문제를 해결할 수 있다. 원반이 2 개인 경우, 0 번 원반을 B 에 옮기고, 1 번 원반을 C 에 옮긴 뒤, B 에 있는 0 번 원반을 C 에 옮기는 것으로 문제를 해결할 수 있다.
이런식으로 원반의 갯수를 늘려갈 때, 문제는 이전의 문제를 포함하고 있는 확장된 문제라고 정의할 수 있다. 그리고 확장된 문제의 경우, 이전 문제에서 사용된 원반보다 큰 원반이 추가된 경우 이므로, 이전 문제에서 해결된 방법이 그대로 적용될 수 있게 된다.
즉, 확장된 문제는 포함하고 있는 이전 문제를 해결한 뒤 (단 spare 와 destination 의 위치는 바뀌어야 한다.), 추가된 원반을 destination 으로 옮기고, 다시 이전 문제를 풀면 해결된다. 이보다 확장된 경우에도, 현재 문제를 1 번 푼 뒤, 추가된 원반을 옮기고, 현재 문제를 1 번 더 푸는 방법으로 해결이 가능하다. 이는 다음과 같은 점화식으로 표현가능하다.

f(n) = 2*f(n-1) + 1
= 2*2*f(n-2) + 2 + 1
= 2*2*2*f(n-3) 4 + 2 + 1 //f(1) = 1
= 2^{n-1} + sum{2^{n-2}}
= 2^{n-1} + 2{n-1} -1
 = 2^{n} - 1

채점기준:
1) 확장된 문제가 이전 문제를 포함하고 있음을 명시하였는가.
2) 증명을 위해 귀납적인 방법을 사용하고 있는가.

1)이 있을 경우 정답, 2)가 있을 경우 부분 점수

6. Spanning tree란, 어떤 그래프에서 edge를 적절히 골라내어 만든, 모든 정점을 포함하는 tree를 말한다. 이 때, 그래프 자체가 tree가 아니라면, 어떤 그래프의 spanning tree의 숫자는 여러 개일 수 있다. 이러한 spanning tree 중에, edge의 비용의 합이 최소가 되는 tree를 Minimum Spanning tree라고 부른다.



Line	Weight
(A B)	7
(A C)	5
(A D)	3
(B C)	6
(B D)	10
(B E)	1
(C E)	4
(D E)	6

MST 를 구하는 방법 중에는 아래와 같은 Kruskal's algorithm 이라는 것이 있다.

Kruskal's algorithm

1. T (the final spanning tree) is defined to be the empty set;

2. For each vertex v of G, make the empty set out of v;

3. Sort the edges of G in increasing order;

4. For each edge (u, v) from the sorted list of step 3.
If u and v belong to different sets
Add (u,v) to T;
Get together u and v in one single set;

5. Return T

1) 주어진 그래프의 MST(Minimum spanning tree)를 Kruskal's algorithm 을 사용하여 구하시오.
(MST 를 찾는 과정을 모두 표시하여야 한다)

1) (B E)

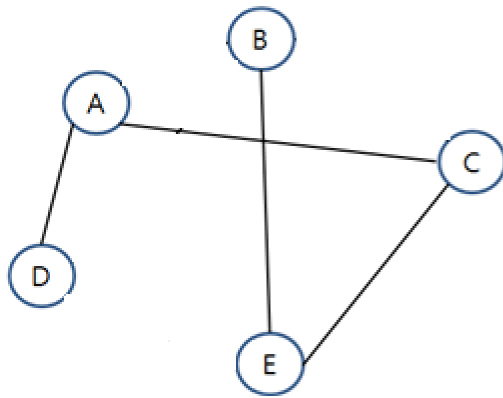
2) ((B E) (A D))

3) ((B E) (A D) (C E))

4) ((B E) (A D) (C E) (A C))

그림으로 그려도 무방하다.

최종 MST



2) Kruskal's algorithm 이 항상 MST 를 찾을 수 있는 지를 증명하시오.

1. Spanning tree 를 만든다.
- T 에 포함된 edge 의 개수는 n-1 보다 같거나 작다. (node 의 개수와 같거나 더 많으면 반드시 cycle 이 생긴다.)
- weight 가 적은 edge 부터 cycle 을 만들지 않는 모든 edge 를 추가하므로, 연결되지 않은 node 가 있을 수 없다.

2. Minimum tree 가 된다.
- 정답이 하나라고 가정. (즉, 같은 비용인데 다른 edge 를 가지고있는 정답이 없음.)
- 정답 tree 를 만드는 edge 의 집합을 Tmst 라고 하면, 알고리즘의 모든 단계에서 T ⊂ Tmst 가 된다.
- 따라서 알고리즘이 종료되면 T = Tmst 가 된다. (어떤 tree 에 속한 tree 인데 edge 의 수가 같다면 두 tree 는 동일한 tree 이다.)

* 2 번의 "알고리즘의 모든 단계에서 T ⊂ Tmst 가 된다."를 증명하기 위해, 수학적 귀납법 사용
base) 최초에 T 는 공집합이다.
step) 알고리즘이 T ⊂ Tmst 인 T 에 edge 인 e 를 추가해서 T'가 됐다고 하자. (T' = T ∪ {e})
이 때 T' ⊂ Tmst 가 된다.

* step 의 " T' ⊂ Tmst "를 증명하기 위해, 귀류법 사용.
- T'가 Tmst 에 포함되지 않는다면, e 또한 Tmst 에 포함되지 않는다.
(T 와 T'의 차이는 e 뿐이므로)
- Tmst ∪ {e} 는 cycle 을 가진다.
(왜냐하면 Tmst 에 e 를 추가하면 총 n 개가 되기 때문이다. n 개의 node 에 n 개의 edge 를 가진 그래프는 cycle 이 존재한다.)
- 그 cycle 은 e 를 포함하고 있지만, 그 cycle 의 모든 edge 가 T 에 포함된 것은 아니다. (e 는 알고리즘이 선택한 edge 이기 때문에.)
- 만들어진 cycle 중에서 T 에 포함되지 않는 edge 를 f 라 하자.
그리고 Tmst 에서 f 를 제거하고 e 를 추가한 집합을 T"이라 하자.
- f 는 e 보다 weight 가 더 크다.. 왜냐하면 알고리즘이 T 에 e 를 추가했기 때문이다.(알고리즘은 최소 weight 를 가진 edge 를 추가한다.)
- 따라서 T"은 Tmst 보다 더 작은 weight 를 보유한 정답이다. 하지만 Tmst 는 유일한 정답이다. 따라서 모순이 생겼다.

모순이 생겼으므로 가정은 거짓이다.
그래서 step 의 T' ⊂ Tmst 는 참이다.

7. Longest Common String(LCS)은 두 문자열의 공통된 문자열 중 가장 긴 문자열을 의미한다. 공통문자열에서 문자의 순서와 문자의 종류는 같아야 하지만, 중간에 다른 문자가 있는 것은 상관하지 않는다.

```
S1: abadbacb
S2: babddcb
LCS(S1, S2): babcb
```

다음은 이러한 LCS 의 길이를 구하기 위한 점화식이다. 여기서 x 와 y 는 문자열의 주어진 위치에 대한 문자 (letter)를 의미하며, i 와 j 는 해당 위치 (index)를 의미한다. 각 문자열의 처음부터 i 와 j 까지를 부분 문자열이라고 보았을 때, $length_{i,j}$는 부분 문자열들 사이에 존재하는 가장 긴 공통 문자열의 길이를 의미한다.

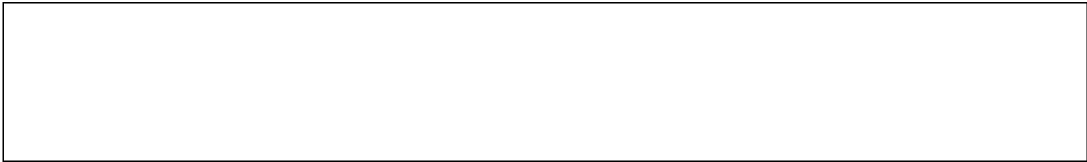
$$length_{i,j}\begin{cases} 0\ (if\ i = 0\ or\ j = 0) \\ length_{i-1,j-1} + 1\ (if\ i,j > 0\ and\ x_i = y_j) \\ \max(length_{i-1,j}, length_{i,j-1})\ (if\ i,j > 0\ and\ x_i \neq y_j) \end{cases}$$

1) LCS 의 길이를 구하기 위한 함수를 재귀적인 방법으로 작성하시오. 단, 괄호와 같은 표현을 제외한 함수의 길이는 6 줄 이하로 작성해야 한다.

```
Int LCS(S1, S2, length1, length2)

If (length1 == 0 || length2 == 0)

   Return 0;

Else if (s1[length1] == s2[length2])

   Return LCS(S1, S2, length1 – 1, length2 – 1) + 1;

Else

   Return max(LCS(S1, S2, length1-1, length2), LCS(S1, S2, length1, length2-1));
```

2) 재귀적인 방법으로 풀었을 때 발생할 수 있는 문제점에 대해서 논하시오.

> 문제의 크기가 커질 경우, 중복된 함수호출이 많아지게 된다.
>
> 기하급수적으로 커지게 되므로, 일정 이상 크기의 문자열을 대상으로 함수를 호출했을 경우, 수행 시간 및 필요한 메모리 공간이 압도적으로 커지게 된다.



3) 재귀적인 방법으로 작성된 함수를 동적 프로그래밍 방법으로 재작성할 때 어떤 부분을 어떻게 고쳐야하는지 1)에서 작성한 함수를 바탕으로 설명하시오. 설명에는 반드시 (1) 추가로 필요한 메모리 공간, (2) 이를 통해 얻을 수 있는 이점 에 대한 설명이 포함되어야 한다.

재귀적으로 작성된 함수는 top-down 방식을 사용한다. 그러므로 중복된 호출이 발생할 수 있다.
이에 대해서 bottom-up 방식으로 바꾸어 풀며, 중간중간의 결과를 배열에 저장하고, 이후에 이 값을 활용하면 중복 함수 호출을 방지할 수 있게 된다.
배열을 만들기 위해서, (length1 + 1) * (length2 + 1) 만큼의 추가 메모리 공간이 필요하다.
재귀적으로 호출되는 부분은 for 문과 같은 loop 문으로 바꾸고, 문자열의 길이가 0 일때부터 length1 과 length2 일 때까지 loop 안에서 반복해서 연산을 수행하도록 고치면 된다.