JooHyun – Lee (comkiwer)

TS주거지검색서비스

Hancom Education Co. Ltd.

Problem

Problem

TS주거지검색서비스

이 나라에는 N 개의 도시가 있다.

N 개의 도시는 3개의 중심 도시와 N-3 개의 주거 도시로 구분된다.

N 개의 도시는 ID 값을 가지며, 그 중 N-3 개의 주거 도시들은 주택가격 값을 가진다.

이 나라에는 지하철 노선이 존재한다.

지하철 노선은 정차 역(=도시)과 역간 거리로 이루어져 있다.

이 지하철 노선은 도시들을 서로 연결한다.

두 도시 사이의 **거리**는 지하철을 통해 갈 수 있는 최단 거리 경로를 의미한다.

단, 여러 개의 노선을 이용 할 경우 환승에 소요되는 거리는 없다.

고객들은 3개의 중심 도시 중 1개 이상의 중심 도시로 출퇴근 한다.

고객이 출근하는 중심 도시의 정보가 주어졌을 때,
그 고객이 살기 적합한 주거 도시를 검색하여 추천해주는 서비스를 만들어보자.

Problem

TS주거지검색서비스

고객에게 적합한 주거 도시를 선정하는 기준은 다음과 같다.

항목은 5가지이며, 1)번 항목이 가장 우선순위가 높고, 5)번 항목이 가장 낮다.

1) 중심 도시 3개는 추천 목록에서 제외한다.
2) "**출근 거리**"의 합이 "**한계 거리**" 보다 큰 경우 추천 목록에서 제외한다.
   ※ 출근 거리는 어떤 도시에서 출근하는 중심도시까지의 거리이다.
3) 주택가격이 가장 낮은 도시가 우선한다.
4) 출근 거리의 합을 구한다. 그 합이 가장 낮은 도시가 우선한다.
5) 도시의 ID 값이 작은 도시가 우선한다.

※ 아래 API 설명을 참조하여 각 함수를 구현하라.

Problem

void init(int N, int mDownTown[])

각 테스트 케이스의 처음에 호출된다.

N 은 도시의 수이다.

각 도시의 ID 값은 1부터 N까지 부여된다.

mDownTown 에는 3 개의 중심 도시 ID 가 저장되어 있다.

이 함수가 호출된 상태에서 모든 도시의 주택가격은 0 이고, 지하철 노선은 없다.

Parameters

N: 도시의 수 (10 ≤ N ≤ 3,000)

mDownTown: 3 개의 중심 도시 ID

Problem

void newLine(int M, int mCityIDs[], int mDistances[])

M 개의 정차 역을 가진 새로운 지하철 노선이 추가된다.

mCityIDs 에는 M 개의 정차 역(=도시)의 ID 값이 순서대로 저장되어 있다.

mCityIDs 에 저장된 도시의 ID 값은 서로 다르다.

mDistances 에는 M-1 개의 수가 순서대로 저장되어 있다.

mDistances[i] 는 도시 mCityIDs[i] 에서 mCityIDs[i+1] 까지의 거리를 의미한다.

두 도시를 연결하는 지하철 노선이 2개 이상일 경우,
두 도시 사이의 거리는 그 노선들의 거리 값 중 가장 작은 값이다.

이 함수가 호출된 직후에 changeLimitDistance() 함수가 호출됨이 보장된다.

Parameters

M: 새로운 지하철 노선의 길이 ID ($2 \le M \le N$)

mCityIDs: 정차하는 도시 목록

mDistances: 각 도시 사이의 거리 ($1 \le mDistances[i] \le 9$) 단, ($0 \le i \le M-2$)

Problem

void changeLimitDistance(int mLimitDistance)

한계 거리를 `mLimitDistance` 로 변경한다.

이 한계 거리는 본문에 설명한 도시선정기준의 2)번 항목에 해당하는 값이다.

이 함수는 `newLine()` 함수가 호출된 직후에만 호출된다.

Parameter

`mLimitDistance`: 새로운 한계 거리

Problem

TS주거지검색서비스

int findCity(int mOpt, int mDestinations[])

고객이 출근하는 중심 도시 정보가 주어졌을 때,
한계 거리 이내에 위치한 주거 도시 들 중 가장 적합한 도시를 추천한다.

`mOpt`는 출근할 중심 도시의 개수이다.

`mDestinations` 는 출근할 중심 도시의 `ID` 값이다. 이 값은 `init()` 에서 전달된 `mDownTown` 의 부분집합이다.

추천된 도시의 주택가격이 1 증가한다.

추천할 도시를 선정하는 기준은 본문 설명을 참고하여라.

Parameters

`mOpt`: 고객이 출근 할 중심 도시의 개수 (1 ≤ mOpt ≤ 3)

`mDestinations`: 중심 도시 `ID` 값

Return

추천할 도시의 `ID`. **단, 한계 거리 이내에 위치한 도시가 하나도 없을 경우 -1**

Problem

TS주거지검색서비스

[제약사항]

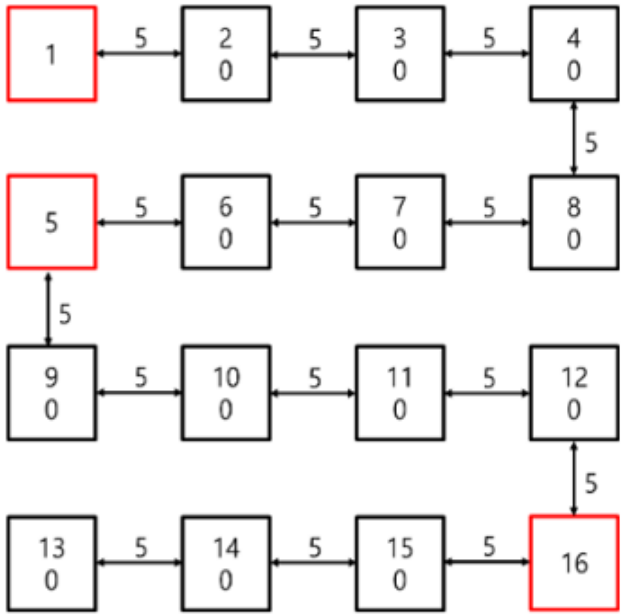
1. **각 테스트 케이스 시작 시** `init()` **함수가 호출된다.**
2. **도시의 수** `N` **은 최대** `3,000` **이다.**
3. **각 테스트 케이스에서** `newLine()` **의 호출 횟수는 최대** `10` **회이다.**
4. `changeLimitDistance()` **함수는** `newLine()` **함수가 호출된 직후에만 호출된다.**
5. **각 테스트 케이스에서** `findCity()` **의 호출 횟수는 최대** `30,000` **회 이다.**

Problem analysis

Problem analysis : 예제

`init()` 이후 `Order #2`의 `newLine()`이 호출된 후의 상태는 `[Fig. 1]` 과 같다.
`16` 개의 도시가 있으며, 그 중 중심 도시는 `1, 5, 16` 이다.
빨간 사각형은 중심 도시를 의미하고, 까만 사각형은 주거 도시를 의미한다.
사각형 안의 첫번째 수는 그 도시의 `ID` 값을 의미하고, 두번째 수는 주택가격을 의미한다.

Order	Function	Return	Figure
1	init(N = 16, mDownTown = {1, 5, 16})		
2	newLine(M = 16, mCityIDs = {1, 2, 3, 4, 8, 7, 6, 5, 9, 10, 11, 12, 16, 15, 14, 13}, mDistances = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5})		[Fig. 1]



[Fig. 1]

Problem analysis : 예제

TS주거지검색서비스

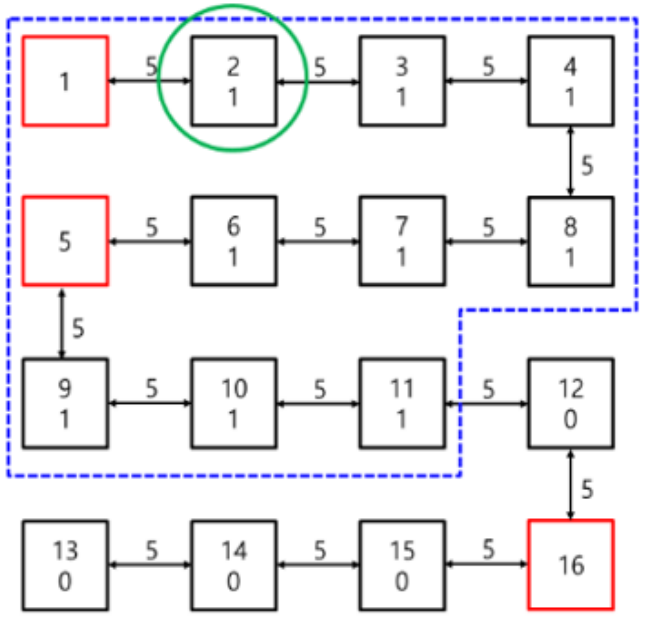
Order #13 의 findCity(2, {1, 5}) 가 호출된 때의 상태는 [Fig. 2] 와 같다.

현재 한계 거리는 65 이고, {1, 5} 2개의 중심 도시와의 출근 거리의 합이 한계 거리 이내인 도시는 {2, 3, 4, 6, 7, 8, 9, 10, 11} 이다.

그 중 출근 거리의 합이 가장 작은 도시는 {2, 3, 4, 6, 7, 8} 이다.

그중 ID 가 가장 작은 2 번 도시를 추천한다.

Order	Function	Return	Figure
3	changeLimitDistance (65)		
4	findCity(mOpt=1,mDestinations={5})	6	
5	findCity(2,{1, 5})	2	
6	findCity(2,{1, 5})	3	
7	findCity(2,{1, 5})	4	
8	findCity(2,{1, 5})	7	
9	findCity(2,{1, 5})	8	
10	findCity(2,{1, 5})	9	
11	findCity(2,{1, 5})	10	
12	findCity(2,{1, 5})	11	
13	findCity(2,{1, 5})	2	[Fig. 2]

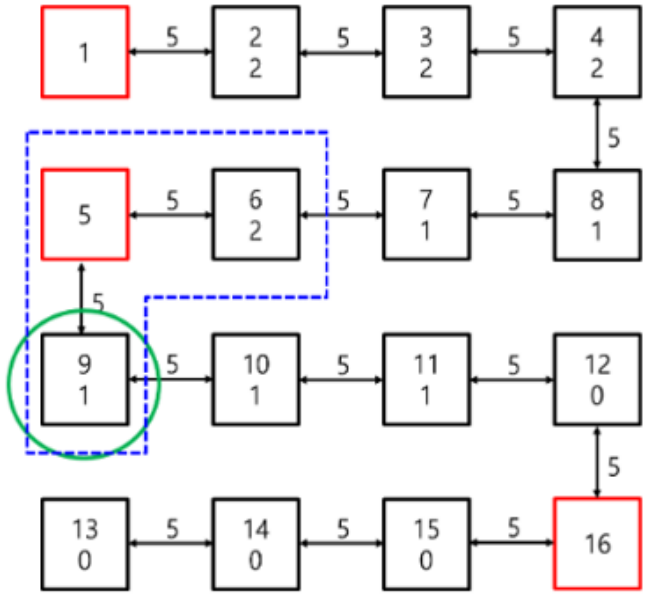


[Fig. 2]

Problem analysis : 예제

Order #17 의 findCity(3, {1, 5, 16}) 가 호출된 때의 상태는 [Fig. 3] 와 같다.
3 개의 중심 도시{1, 5, 16} 와의 거리의 합이 65 이하인 도시는, {6, 9} 2개 이다.
그 중 주택가격이 낮은 9번 도시를 추천한다.

Order	Function	Return	Figure
14	findCity(2,{1, 5})	**3**	
15	findCity(2,{1, 5})	**4**	
16	findCity(2,{1, 5})	**6**	
17	findCity(3,{1, 5, 16})	**9**	**[Fig. 3]**
18	findCity(3,{1, 5, 16})	**6**	

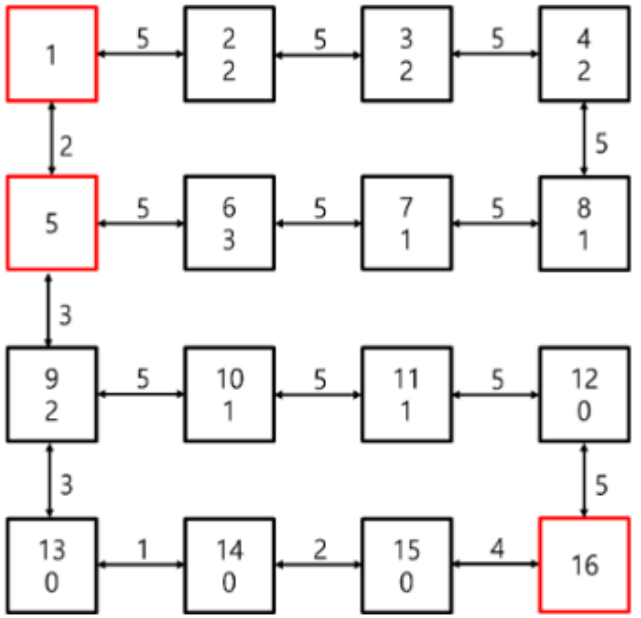


[Fig. 3]

Problem analysis : 예제

Order #19 의 newLine() 이 호출된 이후의 상태는 [Fig. 4] 와 같다.

Order	Function	Return	Figure
19	newLine(M = 7, mCityIDs = {1, 5, 9, 13, 14, 15, 16}, mDistances = {2, 3, 3, 1, 2, 4}		[Fig. 4]
20	changeLimitDistance(45)		
21	findCity(1,{16})	**15**	
22	findCity(1,{5})	**13**	
23	findCity(1,{1})	**14**	
24	findCity(1,{1})	**12**	
25	findCity(1,{1})	**13**	

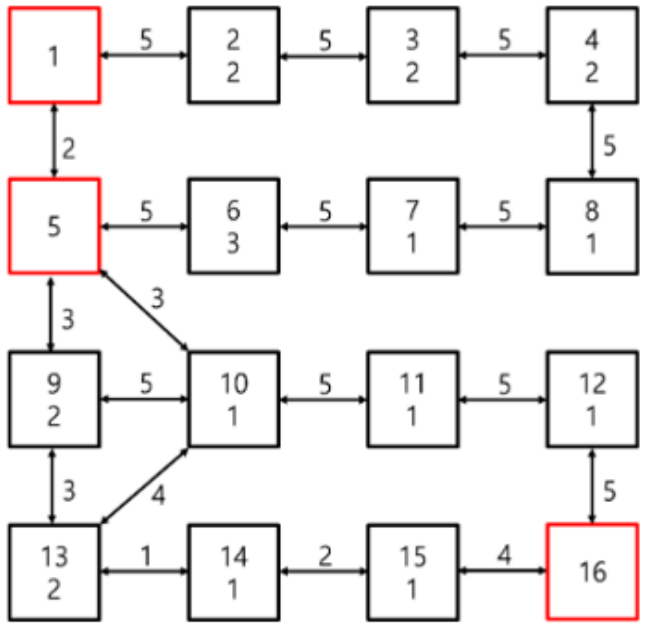


[Fig. 4]

Problem analysis : 예제

Order #26 의 newLine() 이 호출된 이후의 상태는 [Fig. 5] 와 같다.

Order	Function	Return	Figure
26	newLine(M = 4, mCityIDs = {13, 10, 5, 1}, mDistances = {4, 3, 3}		[Fig. 5]
27	changeLimitDistance(35)		
28	findCity(3,{1, 5, 16})	10	
29	findCity(3,{1, 5, 16})	14	
30	findCity(3,{1, 5, 16})	15	
31	findCity(3,{1, 5, 16})	11	
32	findCity(3,{1, 5, 16})	12	
33	findCity(3,{1, 5, 16})	9	
34	findCity(2,{1, 16})	7	
35	findCity(2,{5, 16})	13	
36	findCity(2,{5, 16})	14	

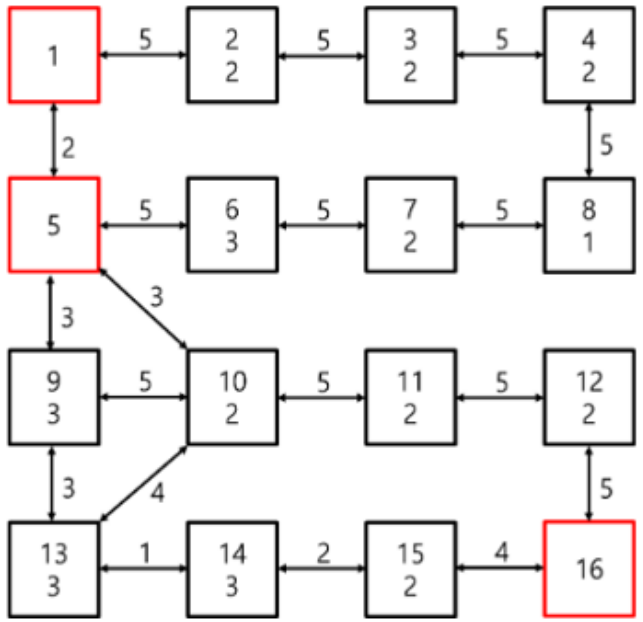


[Fig. 5]

Problem analysis : 예제

테스트케이스 1번이 종료된 이후의 상태는 [Fig. 6] 와 같다.

Order	Function	Return	Figure
26	newLine(M = 4, mCityIDs = {13, 10, 5, 1}, mDistances = {4, 3, 3}		[Fig. 5]
27	changeLimitDistance(35)		
28	findCity(3,{1, 5, 16})	**10**	
29	findCity(3,{1, 5, 16})	**14**	
30	findCity(3,{1, 5, 16})	**15**	
31	findCity(3,{1, 5, 16})	**11**	
32	findCity(3,{1, 5, 16})	**12**	
33	findCity(3,{1, 5, 16})	**9**	
34	findCity(2,{1, 16})	**7**	
35	findCity(2,{5, 16})	**13**	
36	findCity(2,{5, 16})	**14**	



[Fig. 6]

Problem analysis : 제약조건 및 API

TS주거지검색서비스

- 각 도시는 주택가격 속성이 있다.
- 각 도시는 중심도시까지의 최단거리 속성이 있다.
  이는 중심도시로부터 각 도시까지 최단거리와 같다.
- 각 도시는 id 속성이 있다. 1 <= id <= N(3,000)

- void newLine(int M, int mCityIDs[], int mDistances[])함수 호출 뒤에는 반드시
  void changeLimitDistance(int mLimitDistance) 함수가 호출된다.
  따라서 newLine함수 호출이후 필요한 작업중에 어떤 것은
  chaingeLimitDistance() 함수 호출 시에 이루어져야 할 수도 있다.

  newLine() 함수는 노드간의 거리 정보가 최대 M개
  주어지며 최대 10번 호출된다. 2 <= M <= N

Problem analysis : 제약조건 및 API

- int findCity(int mOpt, int mDestinations[])함수의 반환 값으로 채점이 이루어진다.
  int findCity(int mOpt, int mDestinations[])함수는 최대 30,000번 호출된다.

  호출에 대하여 다음 우선순위가 가장 높은 도시 id를 구하고
  주택가격을 1 상승시킨 뒤 id를 반환한다.

  1st. 주택 가격의 asc

  2nd. mOpt개의 중심도시까지 거리합의 asc

  3rd. id의 asc

  우선순위가 가장 높은 도시 id를 구할 수 없는 경우 -1을 반환한다.

Problem analysis : 제약조건 및 API

TS주거지검색서비스

- 하나의 TC에서 findCity() 함수는 최대 30,000 호출되지만
  newLine()함수와 changeLimitDistance()함수는 10번만 호출된다.

- 매 findCity() 함수 호출시마다 중심도시로부터 나머지 모든 도시까지 최단거리를 구하고
  이중에 가장 좋은 도시를 선택하는 것은 시간복잡도가 너무 크다.
  O(30,000:함수호출수 * 3,000(간선수) * 12(노드수의 로그값)*3(중심도시수))

- 함수 newLine() 호출에는 인접 정보만 업데이트하고
  함수 changeLimitDistance() 호출시
  중심도시로부터 나머지 각 도시까지 최단거리를 구하는
  방법을 사용한다면 어떨까?

Solution sketch

Solution sketch

- 함수 newLine() 호출에는 인접 정보만 업데이트하고
  함수 changeLimitDistance() 호출시 한계거리를 업데이트 하고
  중심도시로부터 나머지 각 도시까지 최단거리를 구하는 방법을 사용할 수 있다.
  이때 시간복잡도는
  O(10(함수호출수)*3,000(간선수)*12(노드수의 로그)*3(유형))

- 3개의 중심도시로 만들어지는 7가지 조합에 대하여
  각각 관리한다면 효율적이 될 수 있다.
  세 도시를 0, 1, 2 라고 한다면 findCity() 에서 주어지는 경우는 다음
  0, 1, 2, (0, 1), (0, 2), (1, 2), (0, 1, 2) 7가지가 된다.

- 각 7가지 경우에 대한 최단거리를 set 또는 PQ와 같은 우선순위 자료에 저장한다면
  findCity() 함수 호출에 빠르게 응답할 수 있다.

Solution sketch

- findCity() 호출시 구한 도시가 city라면
  city는 주택가격이 1 상승된다.
  7가지 데이터 집합에서 city 자료를 모두 삭제한 후
  city의 주택가격을 업데이트하고
  7가지 데이터 집합에 city 자료를 새롭게 추가한다.

- 도시의 정보는 다음과 같이 각 속성별 배열로 다룰 수 있다.
  (물론 클래스를 이용할 수도 있다.)

```
const int LM = 3001;
int n;                      // 도시 수
int price[LM];              // 주택가격
int dist[3][LM];            // 각 중심도시로부터의 거리
int D[LM][LM];              // 인접 배열에서 사용되는 두 도시간 거리를 저장.
int types[LM] ;             // 중심도시 index(0, 1, 2)
vector<int> adj[LM];        // 인접 배열에는 도시 정보만 저장.
set<pii> myset[8];          // <(price<<17)+dist, id>
```

Solution sketch

- 중심도시로부터 나머지 모든 도시까지의 거리를 계산하기 위하여 다음과 같은 Dijkstra() 함수를 사용할 수 있다.

```
void Dijkstra(int src, int*drr) {  // O(ElogV) Dijkstra
    for (int i = 1; i <= n; ++i) drr[i] = INF;
    drr[src] = 0;                  // src를 시점으로 최단거리 구하기
    ++vn;
    priority_queue<pii, vector<pii>, greater<pii>> pq; // <int:drr, int:id>
    pq.push({ 0, src });
    while (!pq.empty()) {
        int u = pq.top().second, d = pq.top().first;
        pq.pop();
        if (visited[u] == vn) continue;
        visited[u] = vn;
        for (int v : adj[u]) if (visited[v] < vn) {
            int nd = d + D[u][v];
            if (nd < drr[v]) {
                drr[v] = nd;
                pq.push({ nd, v });
            }
        }
    }
}
```

Solution sketch

- 이제 각 함수별 할 일을 정리해 보자.

void init(int N, int mDownTown[])

```
for (int i = 1; i <= n; ++i) {          // 이전 자료 초기화
    for (int j : adj[i]) D[i][j] = 0; // 인접행렬 초기화(사용된 것만)
    adj[i].clear();                     // 인접배열 초기화
    price[i] = 0;                       // 주택 가격 초기화
}
n = N;                                  // 현재 TC 도시수 초기화
for (int i = 0; i < 3; ++i) {
    hub[i] = mDownTown[i];              // 중심도시 저장
    types[hub[i]] = i;                  // i번 중심도시의 번호
    price[hub[i]] = INF;                // 중심도시 주택가격 최고로(최단거리 목록에서 제외할 목적)
}
```

Solution sketch

void newLine(int M, int mCityIDs[], int mDistances[])

```
for (int i = 0; i < M - 1; ++i) {      // 인접행렬 생성(업데이트)과 인접배열 생성하기
    int s = mCityIDs[i], e = mCityIDs[i + 1];
    if (!D[s][e]) {                                    // 인접행렬 생성 & 인접배열 생성
        adj[s].push_back(e), adj[e].push_back(s);
        D[s][e] = D[e][s] = mDistances[i];
    }
    else D[s][e] = D[e][s] = min(D[s][e], mDistances[i]); // 인접행렬 업데이트
}
```

Solution sketch

void changeLimitDistance(int mLimitDistance)

```
limit = mLimitDistance;          // 한계거리 업데이트

for (int i = 0; i < 3; ++i)      // 각 중심도시를 시점으로 모든 다른 도시까지 최단거리 구하기
    Dijkstra(hub[i], dist[i]);

for (int i = 1; i <= 7; ++i)     // 유형별 set 초기화
    myset[i].clear();

for (int i = 1; i <= n; ++i) { // 각 도시를 set[type]에 <(price[cid]<<17)+dist, cid>로 저장
    if (price[i] == INF) continue;
    int a = dist[0][i], b = dist[1][i], c = dist[2][i];
    for (int j = 1; j <= 7; ++j) {
        int nd = (j & 1)*a + ((j >> 1) & 1)*b +((j >> 2) & 1)*c;
        if(nd <= limit)                  // 한계거리를 만족하는 경우만 추가
            myset[j].insert({ ((UI)price[i]<<17) | nd, i });
    }
}
```

Solution sketch

TS주거지검색서비스

int findCity(int mOpt, int mDestinations[])

```
int k = 0;                              // 주어진 중심도시로 만들어지는 유형
for (int I = 0; I < mOpt; ++i)          // 주어진 중심도시로 만들어지는 유형만들기
    k += 1 << types[mDestinations[i]];

if (myset[k].empty()) return -1;        // 가능한 경우가 없는 경우

int city = myset[k].begin()->second;    // k유형중에 가장 우선순위 높은 도시 ID
int a = dist[0][city], b = dist[1][city], c = dist[2][city]; // 세 중심도시까지의 거리
for (int i = 1; i <= 7; ++i) {          // 1. 주택가격을 인상하므로 각 유형별로 set으로부터 일단 제거
    int nd = (i & 1)*a + ((i >> 1) & 1)*b + ((i >> 2) & 1)*c;
    if (nd <= limit)
        myset[i].erase({ ((UI)price[city] << 17) | nd, city });
}
price[city]++;                          // 2. 주택가격 1인상
for (int i = 1; i <= 7; ++i) {          // 3. 각 유형 별로 set에 다시 등록하기
    int nd = (i & 1)*a + ((i >> 1) & 1)*b + ((i >> 2) & 1)*c;
    if( nd <= limit)                    // 제한거리를 만족하는 경우만 추가 *****
        myset[i].insert({ ((UI)price[city] << 17) | nd, city });
}
return city;
```

Solution sketch

[Summay]

- 문제를 분석하고 필요로 하는 idea를 도출할 수 있다.

- 그래프를 표현할 수 있다.
  : 인접 행렬, 인접 리스트, 인접 배열

- 그래프를 탐색할 수 있다.
  : DFS, BFS

- Dijkstra 최단거리 알고리즘을 문제해결에 사용할 수 있다.

- Set, PQ등 우선순위 자료구조를 문제해결에 사용할 수 있다.

Code example1

Code example1


```cpp
#include <vector>
#include <set>
#include <queue>
using namespace std;

using UI = unsigned int;
using pii = pair<UI, int>; // <UI(가격 거리, id)>
const int LM = 3001;
const int INF = 1 << 20;
int n;                     // 도시 수
int limit;                 // 한계 거리
int hub[3];                // 중심도시
int visited[LM], vn;       // 도시 방문 체크 : Dijkstra() 에서 사용
int price[LM];             // 주택가격
int dist[3][LM];           // 각 중심도시로부터의 거리
int D[LM][LM];             // 인접 배열에서 사용되는 두 도시간 거리
int types[LM];
vector<int> adj[LM];       // 인접 배열
set<pii> myset[8];         // <(price<<17)+dist, id>
```

Code example1

TS주거지검색서비스

```
void init(int N, int mDownTown[]) {
    for (int i = 1; i <= n; ++i) {          // 이전 자료 초기화
        for (int j : adj[i]) D[i][j] = 0; // 인접행렬 초기화(사용된 것만)
        adj[i].clear();                     // 인접배열 초기화
        price[i] = 0;                       // 주택 가격 초기화
    }
    n = N;                                  // 현재 TC 도시수 초기화
    for (int i = 0; i < 3; ++i) {
        hub[i] = mDownTown[i];              // 중심도시 저장
        types[hub[i]] = i;                  // i번 중심도시의 번호
        price[hub[i]] = INF;                // 중시도시 주택가격 최고로(최단거리 목록에서 제외할 목적)
    }
}

void newLine(int M, int mCityIDs[], int mDistances[]) {
    for (int i = 0; i < M - 1; ++i) {       // 인접행렬 생성(업데이트)과 인접배열 생성하기
        int s = mCityIDs[i], e = mCityIDs[i + 1];
        if (!D[s][e]) {                                     // 인접행렬 생성 & 인접배열 생성
            adj[s].push_back(e), adj[e].push_back(s);
            D[s][e] = D[e][s] = mDistances[i];
        }
        else D[s][e] = D[e][s] = min(D[s][e], mDistances[i]); // 인접행렬 업데이트
    }
}
```

Code example1


```
void Dijkstra(int src, int*drr) {                          // O(ElogV) Dijkstra
    for (int i = 1; i <= n; ++i) drr[i] = INF;
    drr[src] = 0;                                          // src를 시점으로 최단거리 구하기
    ++vn;
    priority_queue<pii, vector<pii>, greater<pii>> pq; // <int:drr, int:id>
    pq.push({ 0, src });
    while (!pq.empty()) {
        int u = pq.top().second, d = pq.top().first;
        pq.pop();
        if (visited[u] == vn) continue;
        visited[u] = vn;
        for (int v : adj[u]) if (visited[v] < vn) {
            int nd = d + D[u][v];
            if (nd < drr[v]) {
                drr[v] = nd;
                pq.push({ nd, v });
            }
        }
    }
}
```

Code example1

TS주거지검색서비스

```
void changeLimitDistance(int mLimitDistance) {
    limit = mLimitDistance;
    // 각 중심도시를 시점으로 모든 다른 도시까지 최단거리 구하기
    for (int i = 0; i < 3; ++i) {
        Dijkstra(hub[i], dist[i]);
    }

    for (int i = 1; i <= 7; ++i)          // 유형별 set 초기화
        myset[i].clear();

    for (int i = 1; i <= n; ++i) {        // 각 도시별로 set[type]에 <city_id, type> 저장
        if (price[i] == INF) continue;
        int a = dist[0][i], b = dist[1][i], c = dist[2][i];
        for (int j = 1; j <= 7; ++j) {
            int nd = (j & 1)*a + ((j >> 1) & 1)*b +((j >> 2) & 1)*c;
            if(nd <= limit)               // 제한거리를 만족하는 경우만 추가
                myset[j].insert({ ((UI)price[i]<<17) | nd, i });
        }
    }
}
```

Code example1

TS주거지검색서비스

```
int findCity(int mOpt, int mDestinations[]) {
    int k = 0;
    for (int i = 0; i < mOpt; ++i) {       // 주어진 중심도시로 만들어지는 조합을 계산.
        k += 1 << types[mDestinations[i]];
    }

    if (myset[k].empty()) return -1;
    int city = myset[k].begin()->second;
    int a = dist[0][city], b = dist[1][city], c = dist[2][city];
    for (int i = 1; i <= 7; ++i) {         // 1. 주택가격을 인상하므로 set으로부터 일단 제거
        int nd = (i & 1)*a + ((i >> 1) & 1)*b + ((i >> 2) & 1)*c;
        if (nd <= limit)
            myset[i].erase({ ((UI)price[city] << 17) | nd, city });
    }
    price[city]++;                         // 2. 주택가격 1인상
    for (int i = 1; i <= 7; ++i) {         // 3. set에 다시 등록하기
        int nd = (i & 1)*a + ((i >> 1) & 1)*b + ((i >> 2) & 1)*c;
        if( nd <= limit)                   // 제한거리를 만족하는 경우만 추가 *****
            myset[i].insert({ ((UI)price[city] << 17) | nd, city });
    }
    return city;
}
```

Code example2

Code example2


```
#include <vector>
#include <set>
#include <queue>
using namespace std;

using UI = unsigned int;
using pii = pair<UI, int>; // <UI(가격 거리, id)>
const int LM = 3001;
const int INF = 1 << 20;
int n;                      // 도시 수
int limit;                  // 한계 거리
int hub[3];                 // 중심도시
int visited[LM], vn;        // 도시 방문 체크 : Dijkstra() 에서 사용
int price[LM];              // 주택가격
int dist[3][LM];            // 각 중심도시로부터의 거리
int D[LM][LM];              // 인접 배열에서 사용되는 두 도시간 거리
int types[LM];
vector<int> adj[LM];        // 인접 배열
set<pii> myset[8];          // <(price<<17)+dist, id>
struct Data {
    int u, d;   // u:node, d:dist
}que[LM * 10];
int fr, re;
```

Code example2

TS주거지검색서비스

```
void init(int N, int mDownTown[]) {
    for (int i = 1; i <= n; ++i) {          // 이전 자료 초기화
        for (int j : adj[i]) D[i][j] = 0; // 인접행렬 초기화(사용된 것만)
        adj[i].clear();                     // 인접배열 초기화
        price[i] = 0;                       // 주택 가격 초기화
    }
    n = N;                                  // 현재 TC 도시수 초기화
    for (int i = 0; i < 3; ++i) {
        hub[i] = mDownTown[i];              // 중심도시 저장
        types[hub[i]] = i;                  // i번 중심도시의 번호
        price[hub[i]] = INF;                // 중시도시 주택가격 최고로(최단거리 목록에서 제외할 목적)
    }
}

void newLine(int M, int mCityIDs[], int mDistances[]) {
    for (int i = 0; i < M - 1; ++i) {       // 인접행렬 생성(업데이트)과 인접배열 생성하기
        int s = mCityIDs[i], e = mCityIDs[i + 1];
        if (!D[s][e]) {                                     // 인접행렬 생성 & 인접배열 생성
            adj[s].push_back(e), adj[e].push_back(s);
            D[s][e] = D[e][s] = mDistances[i];
        }
        else D[s][e] = D[e][s] = min(D[s][e], mDistances[i]); // 인접행렬 업데이트
    }
}
```

Code example2


```
void Dijkstra(int src, int*drr) {           // O(ElogV) Dijkstra
    for (int i = 1; i <= n; ++i) drr[i] = INF;
    drr[src] = 0;                           // src를 시점으로 최단거리 구하기
    ++vn;
    fr = re = 0;
    que[re++] = { src, 0 };
    while (fr < re) {
        Data t = que[fr++];
        for (int v : adj[t.u]) {
            if (drr[v] > t.d + D[t.u][v]) {
                drr[v] = t.d + D[t.u][v];
                que[re++] = { v, drr[v] };
            }
        }
    }
}
```

Code example2


```
void changeLimitDistance(int mLimitDistance) {
    limit = mLimitDistance;
    // 각 중심도시를 시점으로 모든 다른 도시까지 최단거리 구하기
    for (int i = 0; i < 3; ++i) {
        Dijkstra(hub[i], dist[i]);
    }

    for (int i = 1; i <= 7; ++i)          // 유형별 set 초기화
        myset[i].clear();

    for (int i = 1; i <= n; ++i) {        // 각 도시별로 set[type]에 <city_id, type> 저장
        if (price[i] == INF) continue;
        int a = dist[0][i], b = dist[1][i], c = dist[2][i];
        for (int j = 1; j <= 7; ++j) {
            int nd = (j & 1)*a + ((j >> 1) & 1)*b + ((j >> 2) & 1)*c;
            if (nd <= limit)                // 제한거리를 만족하는 경우만 추가
                myset[j].insert({ ((UI)price[i] << 17) | nd, i });
        }
    }
}
```

Code example2


```
int findCity(int mOpt, int mDestinations[]) {
    int k = 0;
    for (int i = 0; i < mOpt; ++i) {       // 주어진 중심도시로 만들어지는 조합을 계산.
        k += 1 << types[mDestinations[i]];
    }

    if (myset[k].empty()) return -1;
    int city = myset[k].begin()->second;
    int a = dist[0][city], b = dist[1][city], c = dist[2][city];
    for (int i = 1; i <= 7; ++i) {         // 1. 주택가격을 인상하므로 set으로부터 일단 제거
        int nd = (i & 1)*a + ((i >> 1) & 1)*b + ((i >> 2) & 1)*c;
        if (nd <= limit)
            myset[i].erase({ ((UI)price[city] << 17) | nd, city });
    }
    price[city]++;                         // 2. 주택가격 1인상
    for (int i = 1; i <= 7; ++i) {         // 3. set에 다시 등록하기
        int nd = (i & 1)*a + ((i >> 1) & 1)*b + ((i >> 2) & 1)*c;
        if (nd <= limit)                   // 제한거리를 만족하는 경우만 추가 *****
            myset[i].insert({ ((UI)price[city] << 17) | nd, city });
    }
    return city;
}
```

Thank you.