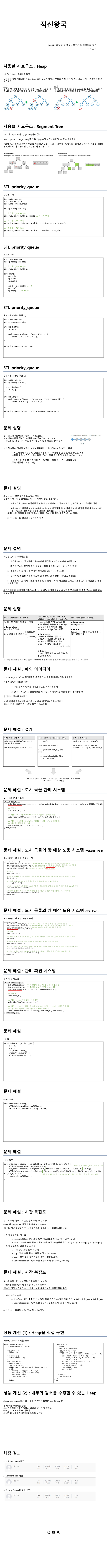
**목차   
1. 문제 풀이 강의 자료   
2. 소스 코드   
   1) PriorityQueue  
   2) SegmentTree**  
  
  
**1. 문제 풀이 강의 자료**  
  
  
**2. 소스 코드**  
**1) PriorityQueue 활용**

**#include < queue >**

**#include < vector >**

**#include < algorithm >**

**#define CITY\_MAX\_N 200**

**#define abs(x) (x >0?x:-(x))**

**using** **namespace** std;

**typedef** pair< **int**, pair< **int**, **int** > > pip;

*// n = 도시 수, m = 관리 수*

**int** n, m;

*// 도시 곡물 관리 시스템*

**struct** CityTaxes {

priority\_queue< pair< **int**, **int** >, vector< pair< **int**, **int** > >, greater< pair< **int**, **int** > > > Q[CITY\_MAX\_N];

*// 초기화*

**void** init() {

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

**while** (!Q[i].empty()) Q[i].pop();

}

}

*// 시간 t에 도시 cityID에 mTax의 곡물이 납부될 예정*

**void** reserveAddTax(**int** cityID, **int** t, **int** mTax) {

Q[cityID].push({ t, mTax });

}

*// 시간 t에 도시 cityID에서 쌓여있는 모든 곡물을 제거 후,*

*// 제거된 곡물의 양 반환*

**int** takeTax(**int** cityID, **int** t) {

**int** taxSum = 0;

**while** (!Q[cityID].empty() && Q[cityID].top().first <= t) {

taxSum += Q[cityID].top().second;

Q[cityID].pop();

}

**return** taxSum;

}

} cityTaxes;

*// 도시 곡물의 양 예상 도움 시스템*

**struct** PredictTaxes {

**int** addedTaxes[CITY\_MAX\_N]; *// addedTaxes[i] : 도시 i에 납부 된 또는 납부 될 곡물 양*

**int** removedTaxes[CITY\_MAX\_N]; *// removedTaxes[i] : 관리에 의해 수도로 이동된 도시 i의 곡물 양*

**bool** isDispatched[CITY\_MAX\_N]; *// isDispatched[i] : 도시 i의 관리 파견 여부*

priority\_queue< pair< **int**, **int** > > Q;

*// 초기화*

**void** init() {

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

addedTaxes[i] = removedTaxes[i] = 0;

isDispatched[i] = **false**;

}

**while** (!Q.empty()) Q.pop();

}

*// 관리가 파견중인 도시는 배제하고,*

*// 관리가 현재 출발해서 곡물이 많을 것으로 예상되는 도시의 ID 반환*

**int** top() {

**while** (!Q.empty() && (isDispatched[-Q.top().second] || Q.top().first != addedTaxes[-Q.top().second] - removedTaxes[-Q.top().second])) Q.pop();

**if** (Q.empty() || Q.top().first == 0) **return** 0;

**return** -Q.top().second;

}

*// 관리를 도시 cityID로 파견*

*// (곡물의 양 예상 대상에서 도시 cityID를 제거)*

**void** pop(**int** cityID) {

isDispatched[cityID] = **true**;

}

*// 도시 cityID로 파견된 관리가 mTax의 곡물을 수도로 납부함*

*// (곡물의 양 예상 대상에서 도시 cityID를 추가)*

**void** push(**int** cityID, **int** mTax) {

isDispatched[cityID] = **false**;

removedTaxes[cityID] += mTax;

Q.push({ addedTaxes[cityID] - removedTaxes[cityID], -cityID });

}

*// 관리가 출발해서 도시 cityID에 도착하였을 때, 예상되는 곡물의 양을 기존에서 mTax 증가시킴*

**void** updatePrediction(**int** cityID, **int** mTax) {

addedTaxes[cityID] += mTax;

**if** (!isDispatched[cityID]) {

Q.push({ addedTaxes[cityID] - removedTaxes[cityID], -cityID });

}

}

} predictTaxes;

*// 관리 파견 시스템*

**struct** OfficialQueue {

**int** officialEmpty; *// 파견되지 않고 대기 중인 관리의 수*

**int** totCapitalTax; *// 수도에 납부된 전체 곡물의 양*

priority\_queue< pip, vector< pip >, greater< pip > > Q;

*// 초기화*

**void** init() {

officialEmpty = m;

totCapitalTax = 0;

**while** (!Q.empty()) Q.pop();

}

*// 시간 tStamp까지 관리 파견 반복*

**void** timeFlow(**int** tStamp) {

**while** (!Q.empty() && Q.top().first <= tStamp) {

**int** t = Q.top().first;

**int** x = Q.top().second.first, y = Q.top().second.second;

Q.pop();

*// 시간 t에 관리 한 명이 도시 x의 곡물 y을 가지고 수도로 돌아옴*

**if** (y >= 0) {

officialEmpty += 1;

totCapitalTax += y;

predictTaxes.push(x, y);

}

*// 시간 t에 관리 한 명이 도시 x에 도착함*

**else** **if** (x >= 0) {

**int** mTax = cityTaxes.takeTax(x, t);

Q.push({ t + x, {x, mTax} });

}

*// 시간 t에 도시 -x의 예측되는 곡물의 양이 -y 만큼 증가함*

**else** {

predictTaxes.updatePrediction(-x, -y);

}

**if** (Q.empty() || Q.top().first > t) {

**while** (officialEmpty > 0 && predictTaxes.top() > 0) {

*// 시간 t에 cityID로 향하는 관리 한 명 출발*

**int** cityID = predictTaxes.top();

predictTaxes.pop(cityID);

Q.push({ t + cityID, {cityID, -1} });

officialEmpty -= 1;

}

}

}

}

*// 시간 tStamp가 되면, 관리가 출발해서 도시 cityID에 도착하였을 때,*

*// 예상되는 곡물의 양을 기존에서 mTax 증가시킴*

**void** updatePrediction(**int** tStamp, **int** cityID, **int** mTax) {

Q.push({ tStamp, {-cityID, -mTax} });

}

} officialQueue;

**void** init(**int** \_n, **int** \_m) {

n = \_n;

m = \_m;

cityTaxes.init();

predictTaxes.init();

officialQueue.init();

}

**void** destroy() {}

**int** check(**int** tStamp) {

officialQueue.timeFlow(tStamp);

**return** officialQueue.totCapitalTax;

}

**int** order(**int** tStamp, **int** cityID\_A, **int** cityID\_B, **int** mTax) {

officialQueue.timeFlow(tStamp - 1);

cityTaxes.reserveAddTax(cityID\_B, tStamp + abs(cityID\_A - cityID\_B), mTax);

officialQueue.updatePrediction(tStamp + max(0, abs(cityID\_A - cityID\_B) - cityID\_B), cityID\_B, mTax);

**return** check(tStamp);

}

**2) Segment Tree 활용**

**#include < queue >**

**#include < vector >**

**#include < algorithm >**

**#define CITY\_MAX\_N 200**

**#define abs(x) (x >0?x:-(x))**

**using** **namespace** std;

**typedef** pair< **int**, pair< **int**, **int** > > pip;

*// n = 도시 수, m = 관리 수*

**int** n, m;

*// 도시 곡물 관리 시스템*

**struct** CityTaxes {

priority\_queue< pair< **int**, **int** >, vector< pair< **int**, **int** > >, greater< pair< **int**, **int** > > > Q[CITY\_MAX\_N];

*// 초기화*

**void** init() {

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

**while** (!Q[i].empty()) Q[i].pop();

}

}

*// 시간 t에 도시 cityID에 mTax의 곡물이 납부될 예정*

**void** reserveAddTax(**int** cityID, **int** t, **int** mTax) {

Q[cityID].push({ t, mTax });

}

*// 시간 t에 도시 cityID에서 쌓여있는 모든 곡물을 제거 후,*

*// 제거된 곡물의 양 반환*

**int** takeTax(**int** cityID, **int** t) {

**int** taxSum = 0;

**while** (!Q[cityID].empty() && Q[cityID].top().first <= t) {

taxSum += Q[cityID].top().second;

Q[cityID].pop();

}

**return** taxSum;

}

} cityTaxes;

*// 도시 곡물의 양 예상 도움 시스템*

**struct** PredictTaxes {

**int** addedTaxes[CITY\_MAX\_N \* 2]; *// addedTaxes[i] : 도시 i에 납부 된 또는 납부 될 곡물 양*

**int** removedTaxes[CITY\_MAX\_N \* 2]; *// removedTaxes[i] : 관리에 의해 수도로 이동된 도시 i의 곡물 양*

**bool** isDispatched[CITY\_MAX\_N \* 2]; *// isDispatched[i] : 도시 i의 관리 파견 여부*

**int** nn;

**int** tree[CITY\_MAX\_N \* 4]; *// 루트노드 인덱스가 1인 세그먼트트리*

**void** updateTree(**int** treeIndex) {

**int** leftTop = tree[treeIndex \* 2], rightTop = tree[treeIndex \* 2 + 1];

**int** predleftTax = addedTaxes[leftTop] - removedTaxes[leftTop], predRightTax = addedTaxes[rightTop] - removedTaxes[rightTop];

tree[treeIndex] = (isDispatched[leftTop] || !isDispatched[rightTop] && predleftTax < predRightTax) ? rightTop : leftTop;

**if** (treeIndex > 1) updateTree(treeIndex / 2);

}

*// 초기화*

**void** init() {

**for** (nn = 1; nn < n; nn \*= 2);

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

addedTaxes[i] = removedTaxes[i] = tree[i] = 0;

tree[nn + i - 1] = i;

isDispatched[i] = **false**;

}

}

*// 관리가 파견중인 도시는 배제하고,*

*// 관리가 현재 출발해서 곡물이 많을 것으로 예상되는 도시의 ID 반환*

**int** top() {

**if** (isDispatched[tree[1]] || addedTaxes[tree[1]] - removedTaxes[tree[1]] <= 0) **return** 0;

**return** tree[1];

}

*// 관리를 도시 cityID로 파견*

*// (곡물의 양 예상 대상에서 도시 cityID를 제거)*

**void** pop(**int** cityID) {

isDispatched[cityID] = **true**;

updateTree((nn + cityID - 1) / 2);

}

*// 도시 cityID로 파견된 관리가 mTax의 곡물을 수도로 납부함*

*// (곡물의 양 예상 대상에서 도시 cityID를 추가)*

**void** push(**int** cityID, **int** mTax) {

isDispatched[cityID] = **false**;

removedTaxes[cityID] += mTax;

updateTree((nn + cityID - 1) / 2);

}

*// 관리가 출발해서 도시 cityID에 도착하였을 때, 예상되는 곡물의 양을 기존에서 mTax 증가시킴*

**void** updatePrediction(**int** cityID, **int** mTax) {

addedTaxes[cityID] += mTax;

updateTree((nn + cityID - 1) / 2);

}

} predictTaxes;

*// 관리 파견 시스템*

**struct** OfficialQueue {

**int** officialEmpty; *// 파견되지 않고 대기 중인 관리의 수*

**int** totCapitalTax; *// 수도에 납부된 전체 곡물의 양*

priority\_queue< pip, vector< pip >, greater< pip > > Q;

*// 초기화*

**void** init() {

officialEmpty = m;

totCapitalTax = 0;

**while** (!Q.empty()) Q.pop();

}

*// 시간 tStamp까지 관리 파견 반복*

**void** timeFlow(**int** tStamp) {

**while** (!Q.empty() && Q.top().first <= tStamp) {

**int** t = Q.top().first;

**int** x = Q.top().second.first, y = Q.top().second.second;

Q.pop();

*// 시간 t에 관리 한 명이 도시 x의 곡물 y을 가지고 수도로 돌아옴*

**if** (y >= 0) {

officialEmpty += 1;

totCapitalTax += y;

predictTaxes.push(x, y);

}

*// 시간 t에 관리 한 명이 도시 x에 도착함*

**else** **if** (x >= 0) {

**int** mTax = cityTaxes.takeTax(x, t);

Q.push({ t + x, {x, mTax} });

}

*// 시간 t에 도시 -x의 예측되는 곡물의 양이 -y 만큼 증가함*

**else** {

predictTaxes.updatePrediction(-x, -y);

}

**if** (Q.empty() || Q.top().first > t) {

**while** (officialEmpty > 0 && predictTaxes.top() > 0) {

*// 시간 t에 cityID로 향하는 관리 한 명 출발*

**int** cityID = predictTaxes.top();

predictTaxes.pop(cityID);

Q.push({ t + cityID, {cityID, -1} });

officialEmpty -= 1;

}

}

}

}

*// 시간 tStamp가 되면, 관리가 출발해서 도시 cityID에 도착하였을 때,*

*// 예상되는 곡물의 양을 기존에서 mTax 증가시킴*

**void** updatePrediction(**int** tStamp, **int** cityID, **int** mTax) {

Q.push({ tStamp, {-cityID, -mTax} });

}

} officialQueue;

**void** init(**int** \_n, **int** \_m) {

n = \_n;

m = \_m;

cityTaxes.init();

predictTaxes.init();

officialQueue.init();

}

**void** destroy() {}

**int** check(**int** tStamp) {

officialQueue.timeFlow(tStamp);

**return** officialQueue.totCapitalTax;

}

**int** order(**int** tStamp, **int** cityID\_A, **int** cityID\_B, **int** mTax) {

officialQueue.timeFlow(tStamp - 1);

cityTaxes.reserveAddTax(cityID\_B, tStamp + abs(cityID\_A - cityID\_B), mTax);

officialQueue.updatePrediction(tStamp + max(0, abs(cityID\_A - cityID\_B) - cityID\_B), cityID\_B, mTax);

**return** check(tStamp);

}