第一题

This problem is equivalent to finding, out of all contiguous subintervals containing at most K+1 distinct breed IDs, the maximal number of cows of a single breed contained within such an interval.  
  
The idea is to sweep down the array of cows, keep tracking of the left and right endpoints of an interval. Each time we increment the right endpoint, we may need to increment the left endpoint by some amount so that the interval contains at most K+1 distinct IDs. Of course, when we do this, we will increment the left endpoint as little as possible. To do this, we just need to keep track of (i) for each breed ID, how many cows of that ID are in the interval and (ii) how many distinct breed IDs have a nonzero number of cows in the interval.  
  
Now, when examining any interval, we need to know the maximal number of cows of a single breed in that interval. One approach is to use a data structure such as a set or priority queue to maintain the maximum. Since at most K+1 IDs are nonzero at any given time, this solution takes O(N log(K)) time.  
  
An even simpler approach involves the observation that during this sweep process, at some point the left endpoint will actually be pointing to the correct breed ID, and at this time the interval will contain as many cows of that ID as possible. In other words, rather than asking "Given an interval, what is the maximal number of cows of a single ID within this interval?", we ask "Given a cow, what is the maximum number of cows of that ID which can be in an interval with that cow as the left endpoint?" This solution takes O(N) time.  
  
Here is Mark Gordon's solution in C++:

#include <iostream>

#include <cstdio>

#include <map>

#include <set>

using namespace std;

int A[100010];

int main() {

freopen("lineup.in", "r", stdin);

freopen("lineup.out", "w", stdout);

int N, K; cin >> N >> K;

for(int i = 0; i < N; i++) {

cin >> A[i];

}

int res = 0;

int nz\_cnt = 0;

map<int, int> cnt;

for(int i = 0, j = 0; i < N; i++) {

int& ci = cnt[A[i]];

if(ci == 0) nz\_cnt++;

ci++;

for(; nz\_cnt > K + 1; j++) {

int& cj = cnt[A[j]];

--cj;

if(cj == 0) nz\_cnt--;

}

res = max(res, ci);

}

cout << res << endl;

return 0;

}

第二题

This problem is somewhat complex, and the algorithm to solve it uses the following three major steps:

1. Flood fill to find the islands. (Both depth-first search, DFS, and breadth-first search, BFS, will work fine here.)

2. Flood fill to find the distances between all pairs of islands. (BFS should be considerably faster than DFS here.)

3. After finding the distances between all pairs of islands, find the minimum distance needed to traverse all islands. (This is a well-known problem that is also known as the Traveling Salesman Problem.) The simplest solution to this would be to try all possible orderings of the islands, but this is far too slow for N = 15. To speed up the algorithm, we can use dynamic programming, with our state consisting of our current location and the subset of islands that we have visited, and the value as the current total distance. This algorithm can be implemented either recursively or iteratively for a complexity of O(N2 x 2N).

The following is a solution using this idea:

#include <algorithm>

#include <cstdio>

#include <cstring>

#include <queue>

using namespace std;

FILE \*fout = fopen ("island.out", "w");

FILE \*fin = fopen ("island.in", "r");

const int INF = 1000000000;

const int dr [] = {-1, 0, 0, 1};

const int dc [] = {0, -1, 1, 0};

const int MAXN = 16;

const int MAXG = 55;

const int MAXS = 70000;

struct loc

{

int row, col, dis;

loc (int r, int c, int d)

{

row = r, col = c, dis = d;

}

};

int N, R, C;

char grid [MAXG][MAXG];

int group [MAXG][MAXG];

int tdist [MAXG][MAXG];

int dist [MAXN][MAXN];

queue <loc> q;

int best [MAXN][MAXS];

int masks [MAXS];

int msize;

int ans;

inline bool comp (int a, int b)

{

return \_\_builtin\_popcount (a) < \_\_builtin\_popcount (b);

}

void floodfill (int r, int c)

{

group [r][c] = N;

for (int k = 0; k < 4; k++)

{

int nr = r + dr [k];

int nc = c + dc [k];

if (grid [nr][nc] == 'X' && group [nr][nc] == -1)

floodfill (nr, nc);

}

}

void solveislands ()

{

memset (group, -1, sizeof (group));

N = 0;

for (int i = 1; i <= R; i++)

for (int j = 1; j <= C; j++)

if (grid [i][j] == 'X' && group [i][j] == -1)

{

floodfill (i, j);

N++;

}

}

void solvedist ()

{

memset (dist, 63, sizeof (dist));

for (int i = 0; i < N; i++)

{

int ir = -1, ic = -1;

bool found = false;

for (int r = 1; r <= R && !found; r++)

for (int c = 1; c <= C && !found; c++)

if (group [r][c] == i)

{

ir = r, ic = c;

found = true;

}

memset (tdist, 63, sizeof (tdist));

q.push (loc (ir, ic, 0));

tdist [ir][ic] = 0;

while (!q.empty ())

{

loc top = q.front ();

q.pop ();

if (group [top.row][top.col] != -1)

{

if (top.dis < dist [i][group [top.row][top.col]])

dist [i][group [top.row][top.col]] = top.dis;

}

for (int k = 0; k < 4; k++)

{

int nr = top.row + dr [k];

int nc = top.col + dc [k];

if (grid [nr][nc] == 'X')

{

if (top.dis < tdist [nr][nc])

{

tdist [nr][nc] = top.dis;

q.push (loc (nr, nc, top.dis));

}

}

else if (grid [nr][nc] == 'S')

{

if (top.dis + 1 < tdist [nr][nc])

{

tdist [nr][nc] = top.dis + 1;

q.push (loc (nr, nc, top.dis + 1));

}

}

}

}

}

}

void solvetsp ()

{

memset (best, 63, sizeof (best));

for (int i = 0; i < N; i++)

best [i][1 << i] = 0;

msize = 0;

for (int m = 1; m < (1 << N); m++)

masks [msize++] = m;

sort (masks, masks + msize, comp);

for (int ind = 0; ind < msize; ind++)

{

int m = masks [ind];

for (int i = 0; i < N; i++)

if (best [i][m] < INF)

{

for (int j = 0; j < N; j++)

if (best [i][m] + dist [i][j] < best [j][m | (1 << j)])

best [j][m | (1 << j)] = best [i][m] + dist [i][j];

}

}

ans = INF;

for (int i = 0; i < N; i++)

if (best [i][(1 << N) - 1] < ans)

ans = best [i][(1 << N) - 1];

}

int main ()

{

memset (grid, '.', sizeof (grid));

fscanf (fin, "%d %d", &R, &C);

for (int i = 1; i <= R; i++)

fscanf (fin, "%s", &grid [i][1]);

solveislands ();

solvedist ();

solvetsp ();

fprintf (fout, "%d\n", (ans < INF) ? ans : -1);

return 0;

}

第三题

Let's start by just considering how to find the location for seating arriving cows. Clearly a linear scan will be too slow, so we need some sort of data structure to accelerate these queries. Let's think about solving the query recursively: if the left half has a big enough gap then we use that, otherwise we try the gap that crosses from the left to the right half (if any), and failing that we try the right half.

Our data structure should match this recursive approach, so we use a binary tree, where the root node represents the entire range of seats and each child of a node represents the left or right half of the parent. We will need to store a few fields in each node:

1. The biggest empty gap in the node
2. The size of the gap adjacent to the left edge
3. The size of the gap adjacent to the right edge

This contains all the information necessary to answer the queries in O(log N) time each.

Next, we need to consider how to apply updates: either seating the party for whom we've just found a gap, or freeing up seats in a range. The first thing to note is that the fields we store in a node can be recomputed from the corresponding fields in the two children. Thus, a simple approach to modifying a node is to make the modifications to the two children (with an early out if the range to update does not intersect both children), and then recomputing the current node. However, that will require time proportional to the length of the range, which will again be too slow. What we need is some way to quickly mark a higher-level node as completely full or completely empty, without visiting all its descendants. In fact, we do exactly that, by adding a field to each node to indicate whether it is completely full, completely empty, or other. When a node is marked completely full/empty, its descendants have undefined values and should not be consulted. When updating a node it will sometimes need to change from completely full/empty to other; in this case the full/empty status needs to be propagated to its children. The query process given above then needs to be slightly modified to process full/empty nodes directly without examining the invalid children.

With these changes, both queries and updates can be processed in O(log N) time, making the entire process O(N + M.log N) time.

#include<stdio.h>  
#include<algorithm>  
  
using std::max;  
  
const int MAXN=524288;  
  
char f[MAXN<<1];//0:empty 1:full 2:mixed  
int ga[MAXN<<1],gl[MAXN<<1],gr[MAXN<<1];  
inline void push(int n)  
{  
int lc=n<<1,rc=n\*2+1;  
if(f[n]==1)  
{  
f[lc]=f[rc]=1;  
ga[lc]=gl[lc]=gr[lc]=ga[rc]=gl[rc]=gr[rc]=0;  
}  
else if(!f[n])  
{  
f[lc]=f[rc]=0;  
ga[lc]=gl[lc]=gr[lc]=(ga[n]+1)>>1;  
ga[rc]=gl[rc]=gr[rc]=ga[n]>>1;  
}  
}  
inline void update(int n)  
{  
int lc=n<<1,rc=n\*2+1;  
if(!f[lc] && !f[rc])  
{  
f[n]=0;  
ga[n]=gl[n]=gr[n]=ga[lc]+ga[rc];  
}  
else if(f[lc]==1 && f[rc]==1)  
{  
f[n]=1;  
ga[n]=gl[n]=gr[n]=0;  
}  
else  
{  
f[n]=2;  
gl[n]=gl[lc];  
if(!f[lc] && f[rc]==2)  
gl[n]+=gl[rc];  
gr[n]=gr[rc];  
if(!f[rc] && f[lc]==2)  
gr[n]+=gr[lc];  
ga[n]=max(gr[lc]+gl[rc],max(ga[lc],ga[rc]));  
}  
}  
int query(int n,int l,int r,int a)  
{  
if(!f[n])  
return l;  
if(f[n]==1)  
return 0;  
int m=(l+r)>>1,lc=n<<1,rc=n\*2+1;  
if(ga[lc]>=a)  
return query(n<<1,l,m,a);  
if(gr[lc]+gl[rc]>=a)  
return m-gr[lc]+1;  
if(ga[rc]>=a)  
return query(n\*2+1,m+1,r,a);  
return 0;  
}  
void set(int n,int l,int r,int a,int b,bool flag)  
{  
if(r<a || l>b)  
return;  
if(a<=l && r<=b)  
{  
f[n]=flag;  
ga[n]=gl[n]=gr[n]=flag?0:r-l+1;  
return;  
}  
push(n);  
int m=(l+r)>>1;  
set(n<<1,l,m,a,b,flag);  
set(n\*2+1,m+1,r,a,b,flag);  
update(n);  
}  
  
int main()  
{  
freopen("seating.in","r",stdin);  
freopen("seating.out","w",stdout);  
char c[9];  
int n,m,i,a,b;  
scanf("%d%d",&n,&m);  
int ans=0;  
ga[1]=gl[1]=gr[1]=n;  
for(i=0;i<m;++i)  
{  
scanf("%s%d",c,&a);  
if(c[0]=='L')  
{  
scanf("%d",&b);  
set(1,1,n,a,b,0);  
}  
else if((b=query(1,1,n,a)))  
set(1,1,n,b,b+a-1,1);  
else ++ans;  
}  
printf("%d\n",ans);  
return 0;  
}

第四题

Among the given points, we want to look for points (x1, y1) and (x2, y2) such that |x1 - x2| <= k-1 and |y1 - y2| <= k-1. To do this we use a plane sweep. Imagine a vertical strip of width k-1, say from x-k+1 to x. As x increases, we maintain the set of points contained in the strip. Each time a new point (x1, y1) enters the strip, we check for points (x2, y2) in the strip with |y1 - y2| <= k-1. We can do this if we maintain the set of points in the strip using a binary search tree (e.g., TreeSet in Java or STL's set in C++).  
  
We also need to delete points when they leave the strip. We could use a priority queue, keyed by x-coordinate, to help us learn when points need to be deleted. Alternatively, we could use lazy deletion, only deleting points from the set when we come across them.  
  
Here is Mark Gordon's solution in C++:

#include <iostream>

#include <vector>

#include <algorithm>

#include <set>

#include <cstdio>

using namespace std;

int main() {

freopen("squares.in", "r", stdin);

freopen("squares.out", "w", stdout);

int N, K; cin >> N >> K;

vector<pair<int, int> > S;

for(int i = 0; i < N; i++) {

int x, y; cin >> x >> y;

S.push\_back(make\_pair(x, y));

}

sort(S.begin(), S.end());

set<pair<int, int> > st;

vector<pair<int, int> > res;

set<pair<int, int> >::iterator ita, itb;

for(int i = 0, j = 0; i < S.size() && res.size() < 2; i++) {

for(; S[j].first + K <= S[i].first; j++) {

st.erase(make\_pair(S[j].second, j));

}

ita = itb = st.insert(make\_pair(S[i].second, i)).first;

if(ita-- != st.begin() && S[i].second < ita->first + K) {

res.push\_back(make\_pair(i, ita->second));

}

if(++itb != st.end() && itb->first < S[i].second + K) {

res.push\_back(make\_pair(i, itb->second));

}

}

long long result = 0;

if(res.size() > 1) {

result = -1;

} else if(res.size() == 1) {

int dx = S[res[0].first].first - S[res[0].second].first;

int dy = S[res[0].first].second - S[res[0].second].second;

if(dx < 0) dx = -dx;

if(dy < 0) dy = -dy;

result = 1ll \* (K - dx) \* (K - dy);

}

cout << result << endl;

}