NWERC 2011
Solutions to the problems

The Jury

Jacobs University Bremen

Statistics

problem	correct/submissions	fastest
E - Please, go first	62/106	25
B - Bird Tree	44/83	34
C - Move to Front	33/171	20
A - Binomial Coefficients	18/135	22
H - Tichu	13/40	115
I - Tracking RFIDs	7/31	88
G - Smoking Gun	3/63	134
D - Piece it Together	3/39	260
J - Train delay	1/9	279
F - Pool construction	0/2	N/A

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`
- Then the last person that isn't his friend: `Ab9AAbbC22`

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`
- Then the last person that isn't his friend: `Ab9AAbbC22`
- And his friends (in this case none)

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`
- Then the last person that isn't his friend: `Ab9AAbbC22`
- And his friends (in this case none)
- Then the next: `Ab9AAbbC22`

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`
- Then the last person that isn't his friend: `Ab9AAbbC22`
- And his friends (in this case none)
- Then the next: `Ab9AAbbC22`
- And his friends: `A9AAbbbC22`

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`
- Then the last person that isn't his friend: `Ab9AAbbC22`
- And his friends (in this case none)
- Then the next: `Ab9AAbbC22`
- And his friends: `A9AAbbbC22`
- And so on. Final order: `9AAAbbbC22`

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`
- Then the last person that isn't his friend: `Ab9AAbbC22`
- And his friends (in this case none)
- Then the next: `Ab9AAbbC22`
- And his friends: `A9AAbbbC22`
- And so on. Final order: `9AAAbbbC22`
- Now you know the final ordering, count the saved time

E - Please, go first

- Sample case 2: `Ab9AAb2bC2`
- The last person in the line will stay the last person
- All his friends line up in front of him: `Ab9AAbbC22`
- Then the last person that isn't his friend: `Ab9AAbbC22`
- And his friends (in this case none)
- Then the next: `Ab9AAbbC22`
- And his friends: `A9AAbbbC22`
- And so on. Final order: `9AAAbbbC22`
- Now you know the final ordering, count the saved time
- Time saved by `X` is number of positions that the last `X` moved forward times the number of `X`s

E - Please, go first (source code)

```
#include <iostream>
#include <vector>
#include <string>
#include <cctype>
using namespace std;

int main () {
  int runs;
  cin >> runs;

  while (runs--) {
    int n;
    string s;
    cin >> n >> s;

    vector<int> cnt(128,0);
    for (int i=0; i<n; i++) cnt[s[i]]++;

    int res = 0, num_used = 0;
    vector<bool> used(128,false);

    for (int i=n-1; i>=0; i--) {
      if (!used[s[i]]) {
        res += (num_used-(n-1-i))*cnt[s[i]];
        num_used += cnt[s[i]];
        used[s[i]] = true;
      }
    }

    cout << 5 * res << endl;
  }
  return 0;
}
```

B - Bird tree

- Root: 1/1, left: $1/(T+1)$, right: $1+1/T$
- If $a/b < 1$, go left, else go right

B - Bird tree

- Root: $1/1$, left: $1/(T+1)$, right:$1+1/T$
- If $a/b < 1$, go left, else go right
- If left, replace $a/b \to (b/a) - 1 = (b-a)/a$

B - Bird tree

- Root: $1/1$, left: $1/(T+1)$, right:$1+1/T$
- If $a/b < 1$, go left, else go right
- If left, replace $a/b \to (b/a) - 1 = (b-a)/a$
- If right, replace $a/b \to 1/(a/b - 1) = b/(a-b)$

B - Bird tree

- Root: $1/1$, left: $1/(T+1)$, right:$1+1/T$
- If $a/b<1$, go left, else go right
- If left, replace $a/b \to (b/a)-1=(b-a)/a$
- If right, replace $a/b \to 1/(a/b-1)=b/(a-b)$
- Proceed with first step

B - Bird tree

- Root: $1/1$, left: $1/(T+1)$, right:$1+1/T$
- If $a/b<1$, go left, else go right
- If left, replace $a/b \rightarrow (b/a)-1=(b-a)/a$
- If right, replace $a/b \rightarrow 1/(a/b-1)=b/(a-b)$
- Proceed with first step
- Stop once you encounter $a/b=1/1$

B - Bird tree (source code)

```
#include <iostream>
#include <string>

using namespace std;

int main () {

  int runs;
  cin>>runs;

  while (runs--) {

    int a,b;
    char c;
    cin >> a >> c >> b;

    while (a>1 || b>1) {
      if (a<b) {
        cout << "L";
        b -= a;
      }
      else {
        cout << "R";
        a -= b;
      }
      swap(a,b);
    }

    cout << endl;
  }

  return 0;
}
```

C - Movie collection

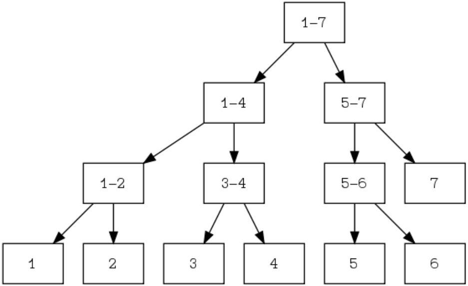
- Note, since $m, r = 100\,000$, you cannot update the stack in $\mathcal{O}(m)$ time

C - Movie collection

- Note, since $m, r = 100\,000$, you cannot update the stack in $\mathcal{O}(m)$ time
- Therefore you need some smart data structure to store information

C - Movie collection

- Note, since $m, r = 100\,000$, you cannot update the stack in $\mathcal{O}(m)$ time
- Therefore you need some smart data structure to store information
- Binary indexed tree/Fenwick tree does the job in $\mathcal{O}(log(m))$ time



- At every step, take movie x and put it back on -1,-2,-3,. . .

C - Movie collection

Note that this is also $O(n^2)$:

```
for (int j = 0; j < r; j++) {
  movie = sc.nextInt();
  index = movies.indexOf(movie);
  System.out.print("" + (m - index - 1) + " ");
  movies.removeElementAt(index);
  movies.add(movie);
}
```

A - Binomial coefficients (1)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$

A - Binomial coefficients (1)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Only look for solutions with $k \leq n/2$ and count twice if necessary

A - Binomial coefficients (1)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Only look for solutions with $k \leq n/2$ and count twice if necessary
- Loop over k from 0 to $\binom{2k}{k} > x$

A - Binomial coefficients (1)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Only look for solutions with $k \leq n/2$ and count twice if necessary
- Loop over k from 0 to $\binom{2k}{k} > x$
- Binary search for n

A - Binomial coefficients (1)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Only look for solutions with $k \leq n/2$ and count twice if necessary
- Loop over k from 0 to $\binom{2k}{k} > x$
- Binary search for n
- Be really careful with overflows!

A - Binomial coefficients (2)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$

A - Binomial coefficients (2)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Again, only look for solutions with $k \leq n/2$ and count twice if necessary

A - Binomial coefficients (2)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Again, only look for solutions with $k \leq n/2$ and count twice if necessary
- For $k = 1$, solution is $\binom{x}{1}$

A - Binomial coefficients (2)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Again, only look for solutions with $k \leq n/2$ and count twice if necessary
- For $k = 1$, solution is $\binom{x}{1}$
- For $k = 2$, solve $x = \binom{n}{2} = \frac{1}{2}n(n-1)$

A - Binomial coefficients (2)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Again, only look for solutions with $k \leq n/2$ and count twice if necessary
- For $k = 1$, solution is $\binom{x}{1}$
- For $k = 2$, solve $x = \binom{n}{2} = \frac{1}{2}n(n-1)$
- For $k \geq 3$, loop over n until $\binom{n}{k} > x$

A - Binomial coefficients (2)

- Find n, k such that $\binom{n}{k} = \frac{n!}{k!(n-k)!} = x$
- Again, only look for solutions with $k \leq n/2$ and count twice if necessary
- For $k = 1$, solution is $\binom{x}{1}$
- For $k = 2$, solve $x = \binom{n}{2} = \frac{1}{2}n(n-1)$
- For $k \geq 3$, loop over n until $\binom{n}{k} > x$
- Again, be really careful with overflows!

H - Tichu (1)

- Greedy solution

H - Tichu (1)

- Greedy solution
- Brute force over two straights

H - Tichu (1)

- Greedy solution
- Brute force over two straights
- Greedily take quads, full houses, trips, pair and singletons from the remaining cards

H - Tichu (1)

- Greedy solution
- Brute force over two straights
- Greedily take quads, full houses, trips, pair and singletons from the remaining cards
- One tricky case: 2 full houses is better than 1 quads + 2 trips

H - Tichu (2)

- Bitmask dynamic programming solution

H - Tichu (2)

- Bitmask dynamic programming solution
- For each subset ($2^{13} = 8192$) determine whether it is a valid combination

H - Tichu (2)

- Bitmask dynamic programming solution
- For each subset ($2^{13} = 8192$) determine whether it is a valid combination
- DP step: $best[x] = best[x\&!y] + 1$ with x, y bitmasks, $x\&y = y$ and y a valid combination

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$
- "`k heard i shoot before j`" leads to $t_{ij} = d_{kj} - d_{ki}$

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$
- "`k heard i shoot before j`" leads to $t_{ij} = d_{kj} - d_{ki}$
- Use Floyd-Warshall to draw inferences: $t_{ij} \leq t_{ik} + t_{kj}$

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$
- "`k heard i shoot before j`" leads to $t_{ij} = d_{kj} - d_{ki}$
- Use Floyd-Warshall to draw inferences: $t_{ij} \leq t_{ik} + t_{kj}$
- This is all information that needs to be obtained

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$
- "`k heard i shoot before j`" leads to $t_{ij} = d_{kj} - d_{ki}$
- Use Floyd-Warshall to draw inferences: $t_{ij} \leq t_{ik} + t_{kj}$
- This is all information that needs to be obtained
- If $t_{ii} < 0$, it is impossible

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$
- "`k heard i shoot before j`" leads to $t_{ij} = d_{kj} - d_{ki}$
- Use Floyd-Warshall to draw inferences: $t_{ij} \leq t_{ik} + t_{kj}$
- This is all information that needs to be obtained
- If $t_{ii} < 0$, it is impossible
- Otherwise, find i such that $t_{ij} < 0$ for all j

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$
- "`k heard i shoot before j`" leads to $t_{ij} = d_{kj} - d_{ki}$
- Use Floyd-Warshall to draw inferences: $t_{ij} \leq t_{ik} + t_{kj}$
- This is all information that needs to be obtained
- If $t_{ii} < 0$, it is impossible
- Otherwise, find i such that $t_{ij} < 0$ for all j
- If multiple, it is unknown

G - Smoking gun

- Calculate t_{ij}, the minimal time difference between i shooting and j shooting: $t_i \leq t_j + t_{ij}$
- "`k heard i shoot before j`" leads to $t_{ij} = d_{kj} - d_{ki}$
- Use Floyd-Warshall to draw inferences: $t_{ij} \leq t_{ik} + t_{kj}$
- This is all information that needs to be obtained
- If $t_{ii} < 0$, it is impossible
- Otherwise, find i such that $t_{ij} < 0$ for all j
- If multiple, it is unknown
- Otherwise, this gives the unique solution

I - Tracking RFIDs

- To determine whether a sensor can see product, calculate the distance between them

I - Tracking RFIDs

- To determine whether a sensor can see product, calculate the distance between them
- Subtract the number of intersecting walls and compare this with r

I - Tracking RFIDs

- To determine whether a sensor can see product, calculate the distance between them
- Subtract the number of intersecting walls and compare this with r
- Problem: you cannot do this for all pairs of sensors and products

I - Tracking RFIDs

- To determine whether a sensor can see product, calculate the distance between them
- Subtract the number of intersecting walls and compare this with *r*
- Problem: you cannot do this for all pairs of sensors and products
- Note: since sensors are separated by at least *r*, only a few sensors can possibly be in range of a product

I - Tracking RFIDs

- To determine whether a sensor can see product, calculate the distance between them
- Subtract the number of intersecting walls and compare this with r
- Problem: you cannot do this for all pairs of sensors and products
- Note: since sensors are separated by at least r, only a few sensors can possibly be in range of a product
- One possible solution: store all sensors in a search tree (e.g. C++'s `set` or Java's `TreeSet`)

I - Tracking RFIDs

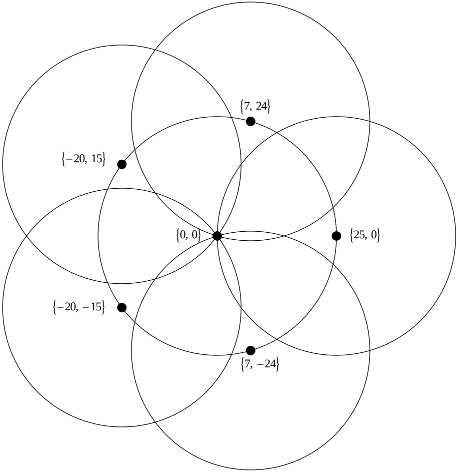
- To determine whether a sensor can see product, calculate the distance between them
- Subtract the number of intersecting walls and compare this with r
- Problem: you cannot do this for all pairs of sensors and products
- Note: since sensors are separated by at least r, only a few sensors can possibly be in range of a product
- One possible solution: store all sensors in a search tree (e.g. C++'s `set` or Java's `TreeSet`)
- For each product, look if a sensor at $(x + \delta x, y + \delta y)$ exists for $-r \leq \delta x, \delta y \leq r$

I - Tracking RFIDs

- To determine whether a sensor can see product, calculate the distance between them
- Subtract the number of intersecting walls and compare this with r
- Problem: you cannot do this for all pairs of sensors and products
- Note: since sensors are separated by at least r, only a few sensors can possibly be in range of a product
- One possible solution: store all sensors in a search tree (e.g. C++'s `set` or Java's `TreeSet`)
- For each product, look if a sensor at $(x + \delta x, y + \delta y)$ exists for $-r \leq \delta x, \delta y \leq r$
- More difficult solutions using binning, quad trees are also possible

I - Tracking RFIDs (test case)

- A maximum of 6 sensors can be in range of a product:



D - Piece it together (1)

- Solution: not matching/max.flow

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack
- Solution: 2SAT

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack
- Solution: 2SAT
- First, check *white* $= 2 \times$ *black*

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack
- Solution: 2SAT
- First, check $white = 2 \times black$
- Boolean variables: x is part of the same puzzle piece as y (x, y adjacent)

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack
- Solution: 2SAT
- First, check $white = 2 \times black$
- Boolean variables: x is part of the same puzzle piece as y (x, y adjacent)
- Black square should be connected to its left xor right white neighbor: $(A|B)\&(!A|!B)$

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack
- Solution: 2SAT
- First, check $white = 2 \times black$
- Boolean variables: x is part of the same puzzle piece as y (x, y adjacent)
- Black square should be connected to its left xor right white neighbor: $(A|B)\&(!A|!B)$
- Identically, it should be connected to its upper or lower neighbor

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack
- Solution: 2SAT
- First, check $white = 2 \times black$
- Boolean variables: x is part of the same puzzle piece as y (x, y adjacent)
- Black square should be connected to its left xor right white neighbor: $(A|B)\&(!A|!B)$
- Identically, it should be connected to its upper or lower neighbor
- White square should be connected to at most one black square: $(!A|!B)\&(!A|!C)\&(!B|!C)$ etc.

D - Piece it together (1)

- Solution: not matching/max.flow
- Solution: not backtrack
- Solution: 2SAT
- First, check $white = 2 \times black$
- Boolean variables: x is part of the same puzzle piece as y (x, y adjacent)
- Black square should be connected to its left xor right white neighbor: $(A|B)\&(!A|!B)$
- Identically, it should be connected to its upper or lower neighbor
- White square should be connected to at most one black square: $(!A|!B)\&(!A|!C)\&(!B|!C)$ etc.
- Now you can use a standard 2SAT solution

D - Piece it together (2)

- But this problem has way more structure!

D - Piece it together (2)

- But this problem has way more structure!
- You can divide the problem in four subproblems

D - Piece it together (2)

- But this problem has way more structure!
- You can divide the problem in four subproblems
- Take white squares at $x = a \mod 2$ and $y = b \mod 2$ $(a, b = 0, 1)$ and adjacent black squares

D - Piece it together (2)

- But this problem has way more structure!
- You can divide the problem in four subproblems
- Take white squares at $x = a \mod 2$ and $y = b \mod 2$ $(a, b = 0, 1)$ and adjacent black squares
- Each connected component of these subproblems should have $white = black$

D - Piece it together (2)

- But this problem has way more structure!
- You can divide the problem in four subproblems
- Take white squares at $x = a \mod 2$ and $y = b \mod 2$ $(a, b = 0, 1)$ and adjacent black squares
- Each connected component of these subproblems should have $white = black$
- If so, it's possible to solve the puzzle, otherwise, it's not

D - Piece it together (source code)

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int Y,X,W,B,py,px;
vector<string> s;
vector<vector<bool> > u;

void go (int y, int x) {

  if (y<0||y>=Y||x<0||x>=X) return;

  if (u[y][x]) return;
  u[y][x]=true;

  if (s[y][x]=='.') return;
  if (s[y][x]=='W' && (y+py)%2+(x+px)%2!=0)
    return;
  if (s[y][x]=='B' && (y+py)%2+(x+px)%2!=1)
    return;

  if (s[y][x]=='W') W++;
  if (s[y][x]=='B') B++;

  go(y-1,x);
  go(y+1,x);
  go(y,x-1);
  go(y,x+1);
}
```

```
int main () {
  int runs;
  cin >> runs;

  while (runs--) {
    cin >> Y >> X;
    s = vector<string>(Y);
    for (int y=0; y<Y; y++)
      cin >> s[y];

    bool ok=true;

    for (px=0; px<2; px++)
      for (py=0; py<2; py++) {
        u = vector<vector<bool> >
            (Y,vector<bool>(X, false));

        for (int y=0; y<Y; y++)
          for (int x=0; x<X; x++) {
            W=B=0;
            go(y,x);
            if (W!=B) ok=false;
          }
      }

    cout << (ok ? "YES" : "NO") << endl;
  }

  return 0;
}
```

J - Train delays (1)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$

J - Train delays (1)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Easiest: use Bellman-Ford algorithm for shortest paths

J - Train delays (1)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Easiest: use Bellman-Ford algorithm for shortest paths
- Initially, $best[end, t] = 0$ and $best[other, t] = \infty$

J - Train delays (1)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Easiest: use Bellman-Ford algorithm for shortest paths
- Initially, $best[end, t] = 0$ and $best[other, t] = \infty$
- Loop over all trains, calculate potential new expected time $best[from, depart]$

J - Train delays (1)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Easiest: use Bellman-Ford algorithm for shortest paths
- Initially, $best[end, t] = 0$ and $best[other, t] = \infty$
- Loop over all trains, calculate potential new expected time $best[from, depart]$
- If it's better, update $best[from, depart]$ and $best[from, t]$ for all t

J - Train delays (1)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Easiest: use Bellman-Ford algorithm for shortest paths
- Initially, $best[end, t] = 0$ and $best[other, t] = \infty$
- Loop over all trains, calculate potential new expected time $best[from, depart]$
- If it's better, update $best[from, depart]$ and $best[from, t]$ for all t
- Repeat until nothing changes anymore

J - Train delays (1)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Easiest: use Bellman-Ford algorithm for shortest paths
- Initially, $best[end, t] = 0$ and $best[other, t] = \infty$
- Loop over all trains, calculate potential new expected time $best[from, depart]$
- If it's better, update $best[from, depart]$ and $best[from, t]$ for all t
- Repeat until nothing changes anymore
- Use epsilon when comparing doubles!

J - Train delays (2)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$

J - Train delays (2)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \dots 59$
- Harder: use Dijkstra's algorithm for shortest paths

J - Train delays (2)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Harder: use Dijkstra's algorithm for shortest paths
- Issue: sometimes you have to update the same state multiple times

J - Train delays (2)

- Calculate best expected time $best[x, t]$ for each station x and time $t = 0 \ldots 59$
- Harder: use Dijkstra's algorithm for shortest paths
- Issue: sometimes you have to update the same state multiple times
- At most 60 times though

F - Pool construction

- Solution: maximum flow

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:
  - Vertices: source, sink, every square

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:
  - Vertices: source, sink, every square
  - Edge from source to boundary square with capacity ∞

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:
  - Vertices: source, sink, every square
  - Edge from source to boundary square with capacity ∞
  - Edge from source to non-boundary grass square with capacity D (dig)

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:
  - Vertices: source, sink, every square
  - Edge from source to boundary square with capacity ∞
  - Edge from source to non-boundary grass square with capacity D (dig)
  - Edge from non-boundary hole square to sink with capacity F (fill)

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:
  - Vertices: source, sink, every square
  - Edge from source to boundary square with capacity ∞
  - Edge from source to non-boundary grass square with capacity D (dig)
  - Edge from non-boundary hole square to sink with capacity F (fill)
  - Edges between connected squares with capacity B (boundary)

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:
  - Vertices: source, sink, every square
  - Edge from source to boundary square with capacity ∞
  - Edge from source to non-boundary grass square with capacity D (dig)
  - Edge from non-boundary hole square to sink with capacity F (fill)
  - Edges between connected squares with capacity B (boundary)
- You can show that the cost of a cut of this graph equals the cost of splitting it into grass and holes along this cut

F - Pool construction

- Solution: maximum flow
- First, fill all boundary squares
- Construct the following flow graph:
  - Vertices: source, sink, every square
  - Edge from source to boundary square with capacity ∞
  - Edge from source to non-boundary grass square with capacity D (dig)
  - Edge from non-boundary hole square to sink with capacity F (fill)
  - Edges between connected squares with capacity B (boundary)
- You can show that the cost of a cut of this graph equals the cost of splitting it into grass and holes along this cut
- So find the minimum cut, i.e., the maximum flow

The end