DP(III)

Theo

Weapon

DP(I)

- State, transition formula
- Bottom-up approach, Top-down approach
- Knapsack (basic)

DP(II)

- Rolling array, bitwise DP
- Tree DP

Other stuff you have learnt

Agenda

DP(III)

- Deeper look into bitwise DP
- Discussion on different kinds of knapsack
- Discussion of different kinds of DP problems

Notice that some of the contents are difficult. You should raise questions whenever you get stucked.

Bitwise DP

Recall: The domino problem (DP(II))

We use a binary number to represent the "shape" of a domino placement in a column. The binary number is called the "bitmask"

Let's discuss some other bitmasks used in other DP problems.

Bitwise DP

Topcoder SRM 655 Div II Hard: NineEasy

For a decimal M of length T (T <= 20) , if we put a binary number "Pattern" with length T to M, then we will get a new decimal in which all the digits with bit positioned there "0" erased.

E.g.
M = 30593, Pattern = 01101, Output = 053
M = 12039, Pattern = 11101, Output = 1209

Bitwise DP

Given N (N <= 5) "Patterns" and T , find the number of M (with leading zeros) so that all outputs of the patterns on M are divisible by 9.

e.g. Pattern = {01, 10}, T = 2,
M = 09, 90, 99, 00 matches the requirement
Therefore answer is 4

Ref:http://community.topcoder.com/stat?c=problem_statement&pm=13720&rd=16415

Bitwise DP

Simple solution: enum all M with |M| = T

O(N*10^T)

will something with O(10^20) pass TL?

Bitwise DP

Observation 1:

A decimal M is divisible by 9 iff

the sum of all digits of M is divisible by 9.

Proof: it is due to the fact that

10^a (a >= 0) % 9 = 1

As 1 % 9 = 1,

10^a % 9 = (10^(a-1) % 9 * 10 % 9) % 9

= (1 * 1) % 9 = 1 by induction

therefore (a_1*10^b+a_2*10^b-1)...) % 9

= (a_1 + a_2...) % 9

Bitwise DP

Observation 1:

A decimal M is divisible by 9 iff

the sum of all digits of M is divisible by 9.

Proof: it is due to the fact that

10^a (a >= 0) % 9 = 1

As 1 % 9 = 1,

10^a % 9 = (10^(a-1) % 9 * 10 % 9) % 9

= (1 * 1) % 9 = 1 by induction

therefore (a_1*10^b+a_2*10^b-1)...) % 9

= (a_1 + a_2...) % 9

Bitwise DP

Let b_1, b_2 ... b_N be the N outputs of the patterns

If we let s_1, s_2 ... s_N be the sums of the digits of the outputs,

Then we are finding all M so that

all s_i % 9 == 0

But the problem is not reduced...

Bitwise DP

Let's construct M from left (most significant) to right (least significant)

When we append a digit to M, say p, for all patterns,
if the corresponding binary is "1", we add p to s_i

Example

M = 428???, Pattern = {100111, 101011, 111100}
Now we add a "5" after 8.

Pattern = 100111	original s = 4	new s = 4 + 1 * 5
Pattern = 101011	original s = 12	new s = 12 + 0 * 5
Pattern = 111100	original s = 14	new s = 14 + 1 * 5

Bitwise DP

Notice that only s % 9 is important, so we can do the modular operation after addition.

Pattern = 100111	original s = 4 % 9 = 4	new s = (4 + 1 * 5) % 9 = 0
Pattern = 101011	original s = 12 % 9 = 3	new s = (3 + 0 * 5) % 9 = 3
Pattern = 111100	original s = 14 % 9 = 5	new s = (5 + 1 * 5) % 9 = 1

Bitwise DP

Notice that only s % 9 is important, so we can do the modular operation after addition.

Pattern = 100111	original s = 4 % 9 = 4	new s = (4 + 1 * 5) % 9 = 0
Pattern = 101011	original s = 12 % 9 = 3	new s = (3 + 0 * 5) % 9 = 3
Pattern = 111100	original s = 14 % 9 = 5	new s = (5 + 1 * 5) % 9 = 1

Bitwise DP

Let's assume we have two unfinished M and M' with length T and K digits inserted, and the resulting s is equal.

e.g. M = 4285??, M' = 0730??
Pattern = {100111, 101011, 111100}
s = {0, 3, 1}

Then, for whatever the remaining "??" is, the final sum array won't differ (because the pattern is constant).

e.g. 428520 == 073020
--> we only need to know the "s" pattern currently && the number of remaining digits to make decision

Bitwise DP

Let S = {s_1, s_2, ... s_N}, Let K be the number of fixed digits,

Then we are finding number of ways to attain
S = {0, 0... 0}, K = |M| = T, and initially:
S = {0, 0... 0}, K = 0

Bitwise DP

State: S and K

transition formula: we are fixing the (K+1) th digit, let's try all 10 of them,

```
handle(S, K)
    u = number_of_ways(S,K)
    for i from 0 to 9
        calculate the new S' if we fix the (K+1) digit to i
        number_of_ways(S', K + 1) += u;
```

as K is strictly increasing, a state will never refer to itself

Bitwise DP

Complexity:
O(number of state) * O(transition per state)

State = {S, K}
We have T choices of K and (9^N) choices of S
({0, 0, 0, 0, 0} ... {8, 8, 8, 8, 8}}

therefore O(number of state) = O(T*9^N)

O(transition per state) = O(A) , where A is the number of choice of digit (= 10)

Complexity = O(T*9^N*10) ~ 20*9^5*10, which is OK

Bitwise DP

How to represent S?

Straightforward:

dp[10][10][10][10][10][30]

dp[a][b][c][d][e][f] -> dp[a+p[1]*i][b+p[2]*i]....

Smart:

dp[9^5+100][30]

Hash the S into a 9-ury number!

Bitwise DP

S = {s_1, s_2 ... s_N}

hash(S) = s_1 + s_2 * 9 + s_3 * 9^2 + ... + s_N * 9 ^ (N - 1)

e.g. S = {0, 3, 1}
f(S) = 0 + 3 * 9 + 1 * 81 = 108

Bitwise DP

```
unhash(Y)
        Let all the element be 0
        Let cnt = 0
        while (Y > 0)
                S[cnt] = Y % 9;
            (extract the last digit and append it to the set)
                cnt++;
                Y /= 9; (remove the last digit)
```

Bitwise DP

e.g. unhash(108)

{0, 0, 0}, Y = 108

as Y % 9 = 0, first element is 0

{0, 0, 0}, Y = 12

as Y % 9 = 3, second element is 3

{0, 3, 0}, Y = 1

as Y % 9 = 1, third element is 1

{0, 3, 1} --> output

Bitwise DP

```
handle(Y, K)
        S = unhash(Y)
    u = number_of_ways(Y,K)
    for i from 0 to 9
        calculate the new S' if we fix the (K+1) digit to i
        number_of_ways(hash(S'), K + 1) += u;
```

Bitwise DP

Usually we use bitwise DP to deal with problems with "column" structure, but in some problems the "column" structure may not be easy to find.

Bitwise DP

Travelling salesman problem:

Given a weighted graph G, find a route to visit every vertex exactly once, with the exception of vertex 1, which is visited exactly twice, and it is the first and last vertex on the route.

1 <= |V| <= 18

Bitwise DP

Brute force: what is the expected complexity?

Consider a complete graph

first vertex: 1 choice

second vertex: |V| - 1 choices

...

last vertex: 1 choice

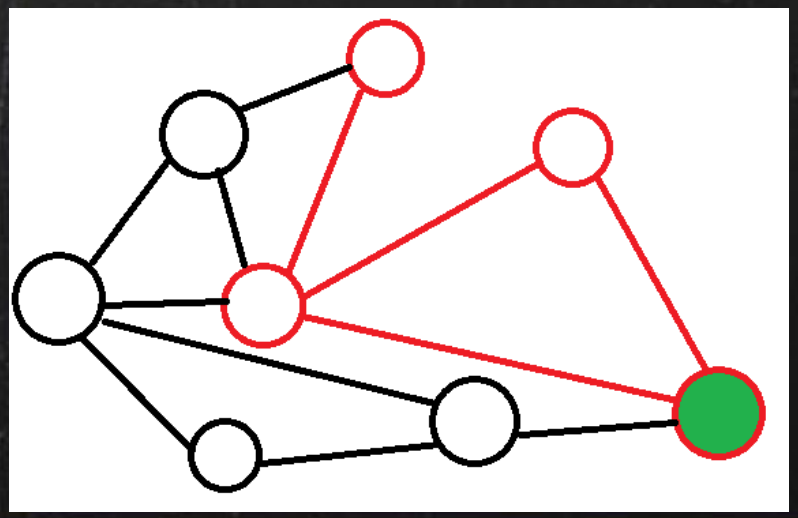
Number of possible decisions is O(|V|!), which exceeds TL

Bitwise DP

Let's ignore the condition that we need to return to the starting city for now.

Assume we have information about the answer for a subset of graph, and the last visited city

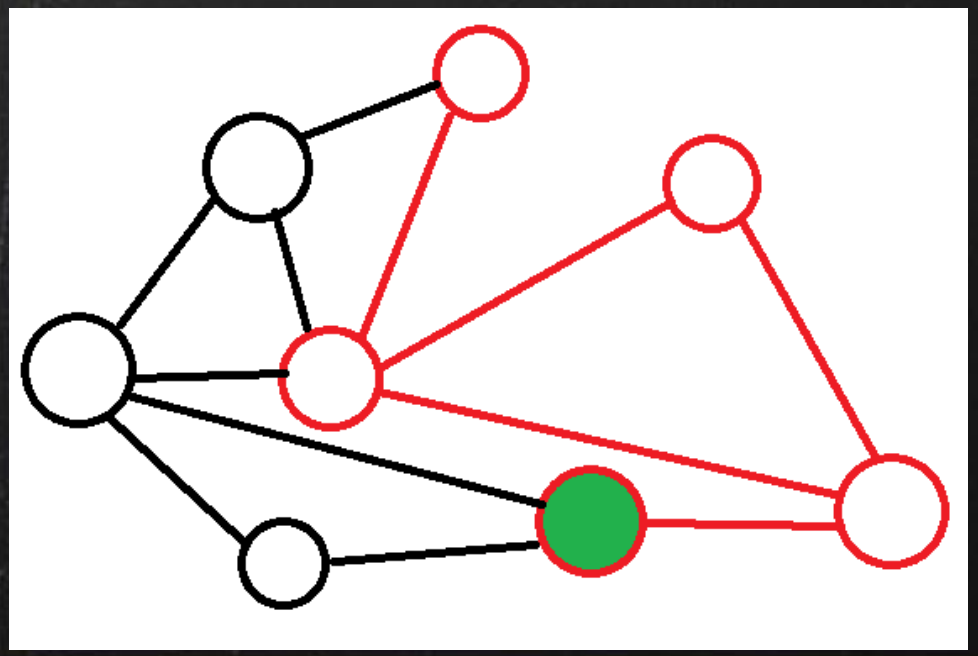
e.g. the red part of the following graph and green vertex as the current city



Bitwise DP

We can immediately transit to new state(s), with a larger subset of node and new last visited city!

Larger: a state will not refer to a smaller state



Bitwise DP

But how can we represent the vertices we have visited?

Hashing again!

Let B = {v_1, v_2, ... v_|V|} be the visited state of the vertices, where 0 is unvisited and 1 is visited.

Then we can represent B as a binary number and translate it to decimal!

Bitwise DP

But afterall we are finding a route, how to solve the circuit version (starting city == ending city)?

First way: use 3-ary number to represent a visiting state, where 0, 1, 2: the remaining number it needs to be visited

O(3^N*N^2), which is not working for N = 18

Bitwise DP

Second way:

We will depart from vertex 1 to neighbours of vertex 1

Let the neighbours be u_1, u_2...

Then initial state will be

S = {u_1 visited, other unvisited}, Cur = u_1

S = {u_2 visited, other unvisited}, Cur = u_2

...

And final state: S = {1, 1, ... 1}, Cur = 1

O(2^|V|*|V|^2), which << O(|V|!)

Agenda

DP(III)

- Deeper look into bitwise DP
- Discussion on different kinds of knapsack
- Discussion of different kinds of DP problems

Notice that some of the contents are difficult. You should raise questions whenever you get stucked.

Knapsack

Recall DP(I) Knapsack problem:

You have N items, the ith item is specified by its two properties v and p:

v: the value you can gain if you buy this item

p: the cost of that item

Now you have a fixed budget C, find the maximal total value T so that the total cost did not exist C.

Knapsack

The basic knapsack is often referred as 0/1 Knapsack.
Today we will discuss:

- Unbounded Knapsack in O(NC)
- Bounded Knapsack in O(NC lg C)
- Conditional Knapsack in O(NC^2)

Unbounded Knapsack

You have N types of items with unlimited supply each, the ith item is specified by its two properties v and p:

v: the value you can gain if you buy this item

p: the cost of that item

Now you have a fixed budget C, find the maximal total value T so that the total cost did not exist C.

Unbounded Knapsack

0/1 Knapsack:

A State (N, C) contributes to

(N + 1, C + p_N) and (N + 1, C)

Unbounded Knapsack:

A State (N, C) contributes to

(N, C + p_N) and (N + 1, C)

As a small state always contributes to a larger state, we can still apply the DP

Unbounded Knapsack

```
for i from 1 to N
        for j from 0 to C
                dp[i + 1][j] = max(dp[i + 1][j], dp[i][j])
                if (j + p_i <= C)
            dp[i][j + p_i] = max(dp[i][j + p_i], dp[i][j] + v_i)
```

Bounded Knapsack

You have N types of items, the ith item is specified by its three properties: v, w and p:

v: the value you can gain if you buy this item

p: the cost of that item

w: the maximal amount of this type of item you can buy

Bounded Knapsack

Simple way:

```
for i from 1 to N
    for j from 0 to C
        for k from 0 to w_i
                dp[i + 1][j] = max(dp[i + 1][j], dp[i][j])
                if (j + p_i * k <= C)
                dp[i + 1][j + p_i * k] =
                    max(dp[i][j + p_i * k], dp[i][j] + v_i * k)
```

That will multiply a $O(C)$ into total complexity.

Bounded Knapsack

Let's assume, for the optimal solution, we pick E (0 <= E <= w) items of certain type to buy.

Let's represent E as a binary number.

Then we buy (0 or 1) 2^0 item,
(0 or 1) 2^1 item,
(0 or 1) 2^2 item...

A 0/1 Selection!

Bounded Knapsack

E.g. if we have a item of
V = 9, P = 23, W = 2

If we buy E = 3 (= 11),
It's equal to we buy a item of V = 9 and W = 2,
and a item of V = 9 * 2 and W = 2 * 2

If we buy E = 14 (= 1110),
It's equal to we buy a item of V = 9 * 2 and W = 2 * 2,
and a item of V = 9 * 4 and W = 2 * 4,
and a item of V = 9 * 8 and W = 2 * 8

Bounded Knapsack

If we can represent P as a set of numbers S so that
- we can compose all numbers from 1 to P by some subset of numbers in S
- we cannot compose any number beyond P by any subset of numbers in S

Then we will get a O(N|S|) 0/1 Knapsack with
total time complexity O(NC|S|)

Bounded Knapsack

Way 1: All "1"s

|S| = O(C),

time complexity = O(NC^2), no improvement

Way 2: Let 2^X be such a number so that 2^X > P

S = {1, 2, 4, ..., 2^(X-1), P - (2^X - 1)}

|S| = O(lg P) = O(lg C)

Bounded Knapsack

But how to prove we can construct all numbers from 1 to P by using S?

1 to 2^X - 1: obvious, by representing this number as binary

2^X to P: Let the required number be Y,
P - (2^X - 1) = Q,
Then we can pick Q, along with Y - Q, which is in [1..2^X-1],

as if Y - Q >= (2^X), then P >= (2^(X+1) - 1), then X should be set to X+1

Bounded Knapsack

For example,

V = 9, P = 21, W = 2

is equal to these set of items:

V = 9 * 1, W = 2 * 1
V = 9 * 2, W = 2 * 2
V = 9 * 4, W = 2 * 4
V = 9 * 8, W = 2 * 8
V = 9 * 6, W = 2 * 6

notice that 1+2+4+8+6 = 21, therefore we cannot pick anything beyond 21

Conditional Knapsack

You have N items, the ith item is specified by its three properties: v, w and p:

v: the value you can gain if you buy this item

p: the cost of that item

f: you must by the f-th item if you buy the ith item

Given: there won't be any cycles formed by the dependence path.

Conditional Knapsack

If we see the items as vertices, and the dependence relationship as edges, Then the dependence relationship forms a forest.

How to transform from a forest to a tree?

Conditional Knapsack

Add a null item I', so that:

the value and cost of I' is 0, and

all items, if not depended by anything else, depends on I'

So the new graph will be a rooted tree with I' as root

Conditional Knapsack

Tree DP: Let (N, C) be the maximal value gained if we have a maximal cost of C, and we are looking at item N.

We will look at the childrens of N one by one.

Conditional Knapsack

Let N have childrens N'

Build up a internal array A, initially 0
Let A[p_N] = c_N (buy this item)

Ask for each child:
for each C' from 0 to C:
"If I give you C' capacity, what is the maximal value?" (query(N', C'))
then use the result to update A table

Lastly we have (N, C) = A[C]

Agenda

DP(III)

- Deeper look into bitwise DP
- Discussion on different kinds of knapsack
- Discussion of different kinds of DP problems

Notice that some of the contents are difficult. You should raise questions whenever you get stucked.

DP problems

CodeChef CHEFGRAD

Given N trees (real one). The ith tree is initially colored R, G or B. Modifying the color of ith tree costs c_i. Find the minimal cost so that all trees of same color is neighbour to each other.

e.g.
N = 8, T = {R, R, R, G, G, R, G, B},
c = {0, 0, 0, 1, 2, 100, 3, 3}

Optimial solution: T = {G, G, G, G, G, R, R, B}, cost = 3
Ref: http://www.codechef.com/problems/CHEFGARD

DP problems

Observation 1:

If we concentrates neighbour trees of same color in the final config, there are only 6 possible configs.

I.e. RGB, RBG, GRB, GBR, BRG, BGR

How to handle cases where one color does not exist?

Pretend it to exist at the end of the tree series.

DP problems

We can focus on one final config, say, RGB

DP(i, j): What is the minimal cost so that the ith tree is colored j?

Notice that j is strictly increasing, we have:

DP(i, j) = DP(i - 1, j) + (T[i] == j?0:c[i])
= DP(i - 1, j - 1) + (T[i] == j?0:c[i])

Final answer for this config =
min(DP(N, 1), DP(N, 2), DP(N, 3))

DP problems

Topcoder SRM 208 Div 1 Hard

You have a (N * M) grid, (N, M <= 50), you start at the left-top corner and move to the bottom-right corner moving bottom and right only, collecting all the apples on the way

After arriving the bottom-right corner, you go back to left-top corner using left and top movement, again collecting all the apples on the way. Apples cannot be collected twice.

DP problems

Then, you move the third and last time, with down and right movement, to bottom-right corner again. we are interested in the maximal number of apples collected.

Ref:http://community.topcoder.com/stat?c=problem_statement&pm=2940&rd=5854

DP problems

How to deal with the 1-path version?

Simple DP:

DP(x, y): maximal possible apple if we walk from (1,1) to (x,y)

transition formula:

DP(x,y) = max(DP(x - 1, y), DP(x, y - 1)) + A[x][y]

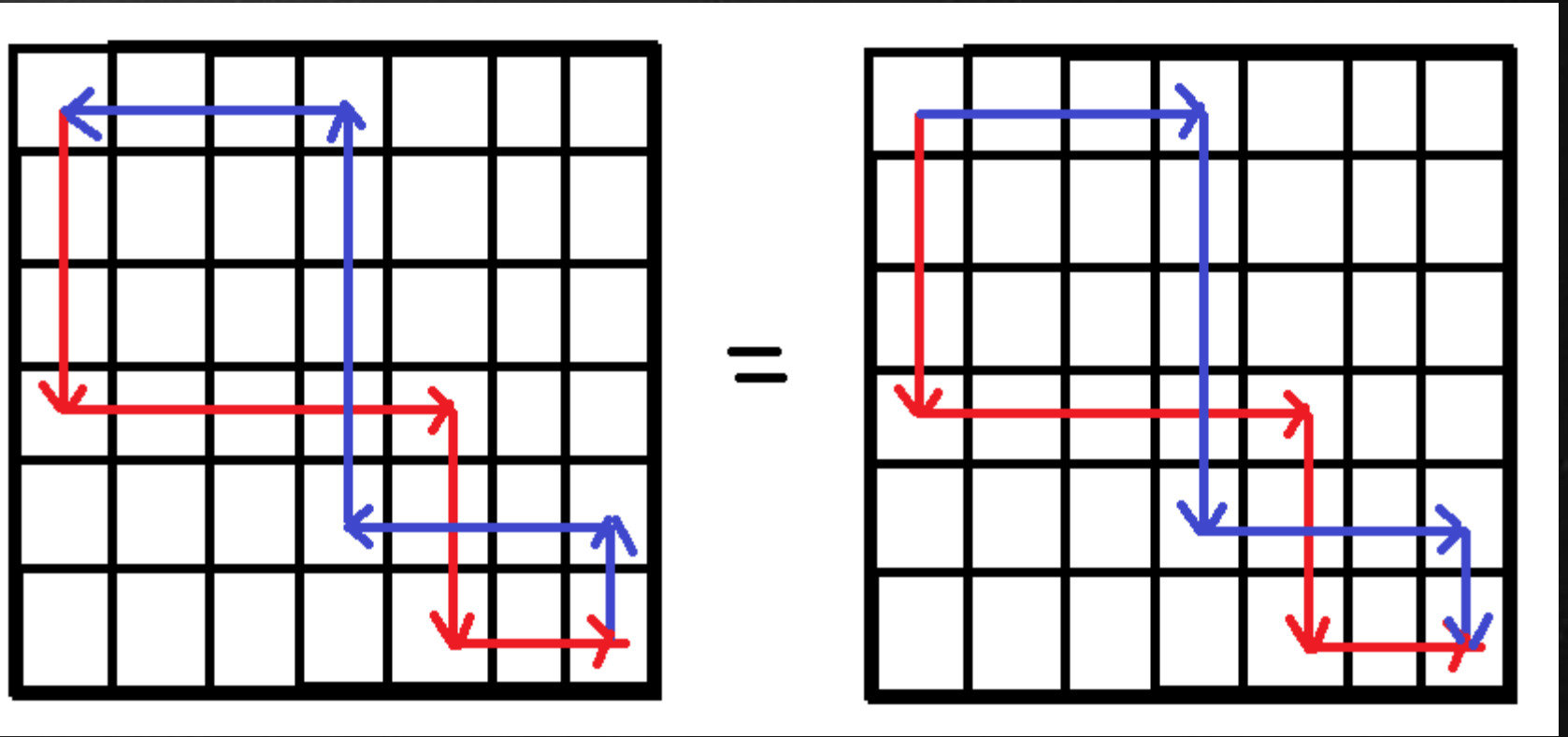
DP problems

How about handling the go-back version?

Problem:
After arriving the bottom-right corner, you go back to left-top corner using left and top movement, again collecting all the apples on the way. Apples cannot be collected twice.

Is Equal to:
Go from top-left to bottom-right corner twice, with right and down movement only.

DP problems



DP problems

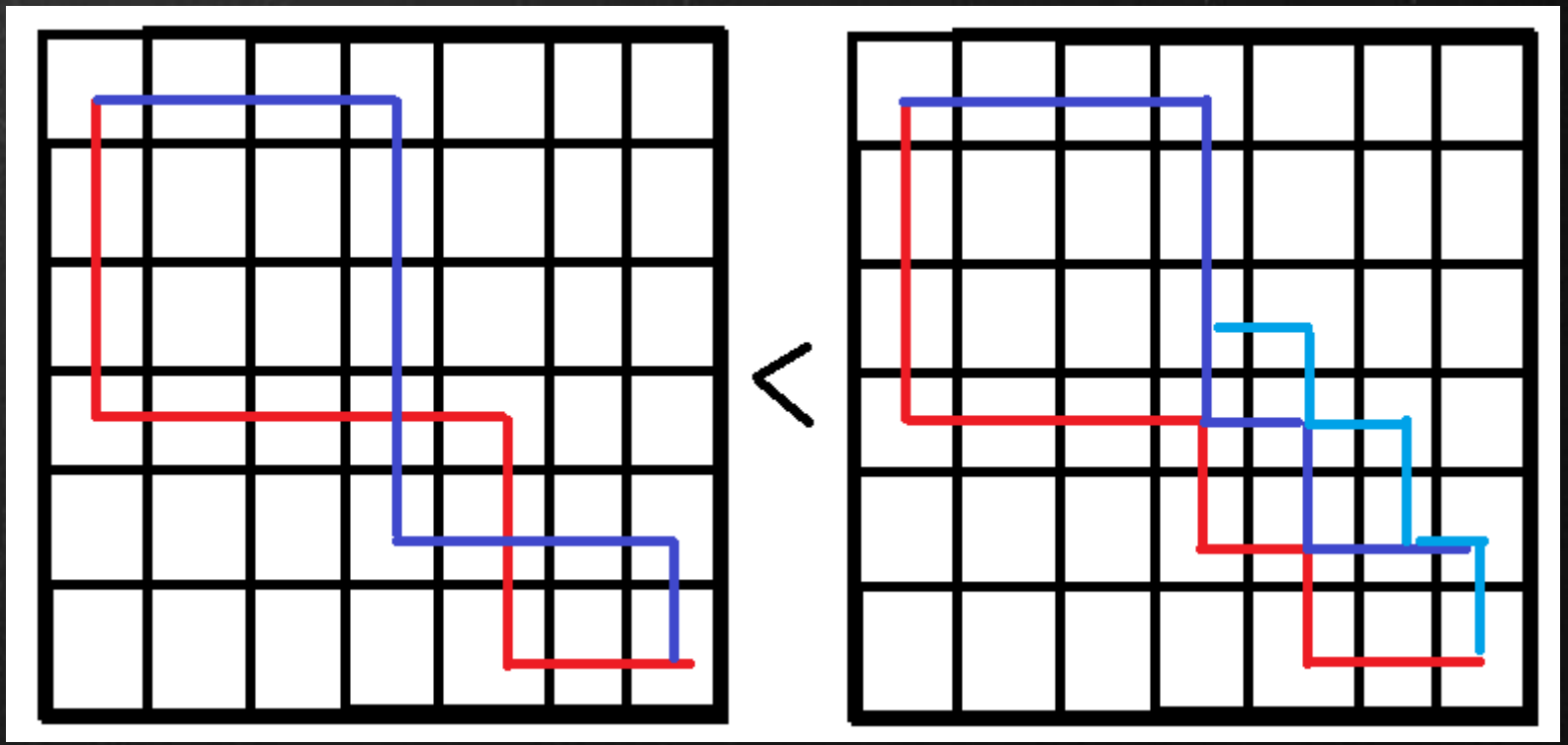
Observation 1:

There exists optimal solution where paths does not intersect.

Reason:

We can resolve the intersection into two paths, then find an alternative path which is more optimal.

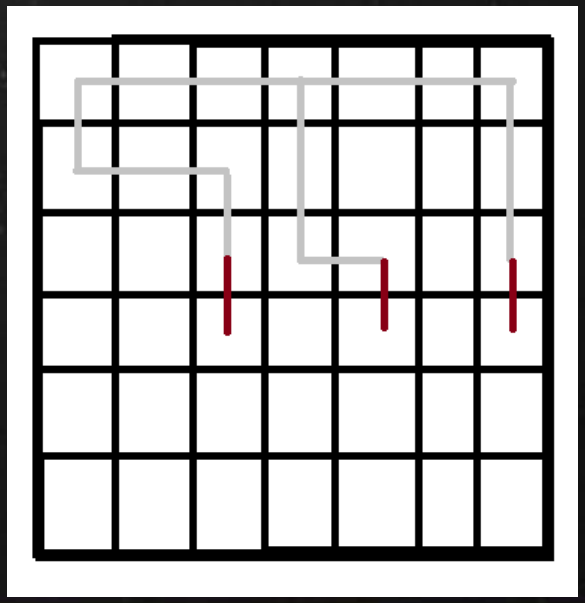
DP problems



DP problems

For the 3-path version, we know that the three paths won't intersect each other at any row R.

Assume we have done the work for previous (R - 1) rows, and we know the maximal value gained so that the 1st path arrives row R at column c_1, 2nd path arrives row R at column c_2, etc.



DP problems

Notice that we do not care about the grey part, as our decision is based on c_1, c_2 and c_3 only.

Let Max(R, c_1, c_2, c_3) be the maximal value.

In row R, we can move some of the paths to the right, so that the path still does not intersect.

DP problems

But how to enum?

We will move the leftmost path (c_1) first, then c_2, afterwards c_3.

To do this, we add an additional counter K, which means that the previous (K - 1) paths are done, and currently looking for kth path.

DP problems

Therefore,

```
DP(R, c_1, c_2, c_3, K)
        DP(R + 1, c_1, c_2, c_3, 1)
        max= DP(R, c_1, c_2, c_3, K) + A(R+1, c_1)
            + A(R+1, c_2) + A(R+1, c_3)
        if (K == 1 && c1 + 1 < c_2)
        DP(R + 1, c_1 + 1, c_2, c_3, 1) max=
                DP(R, c_1, c_2, c_3, K) + A(R, c_1 + 1)
        DP(R + 1, c_1 + 1, c_2, c_3, 2) max=
                DP(R, c_1, c_2, c_3, K) + A(R, c_1 + 1)
        ...
```

DP problems

HKOJ M0421 Two Towers

For a grid, build two square towers on a N * N grid
so that:

- The towers are squares
- Two towers have no point of intersect
- The towers have no point lying on some "invalid" land
- The sum of area of towers is maximal possible

DP problems

One tower version:

DP(x, y) = The largest tower so that the bottom-right corner is (x, y)

= 0 if A(x, y) == 1

= min(DP(x - 1, y - 1), DP(x, y - 1), DP(x - 1, y)) + 1 if A(x,y) == 0

Answer = (Max(DP(x, y))^2)

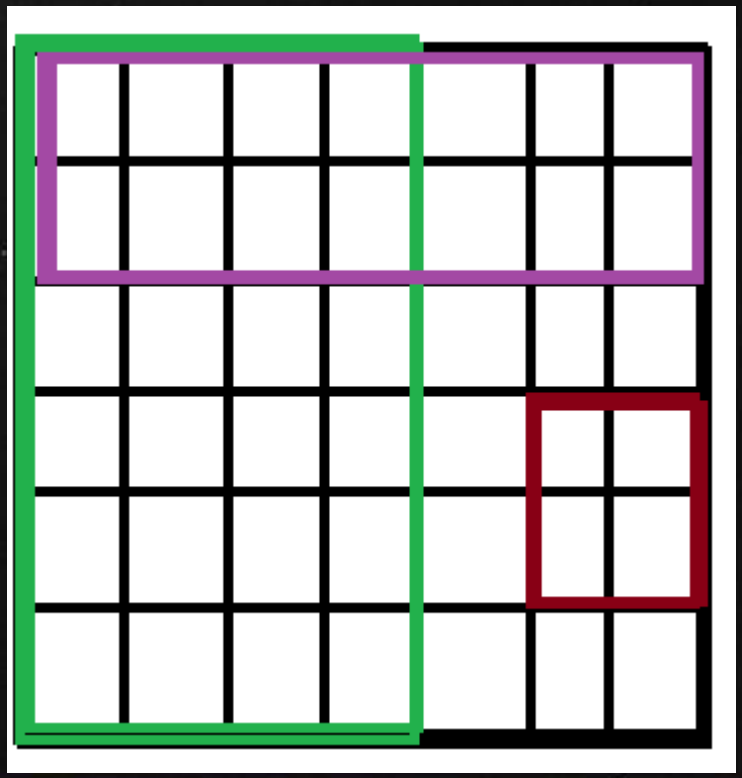
DP problems

How about two towers?

If we fix the first tower, the maximal sum is easy to find:

It will always lie on green region
or purple region.

Handle the cases separately.



DP problems

Let
F(x, y) = The largest tower so that the bottom-right corner is (x, y) and do not cover 2nd tower
= 0 if A(x, y) == 1
= min(min(F(x - 1, y - 1), F(x, y - 1), F(x - 1, y)) + 1,
N-2) if A(x,y) == 0

Then, for the green region,
Let L(y) be max(i from 1..N)(j from 1..y) F(x,y)
similarity,
T(x) = max(i from 1..x)(j from 1..N) F(x,y)

DP problems

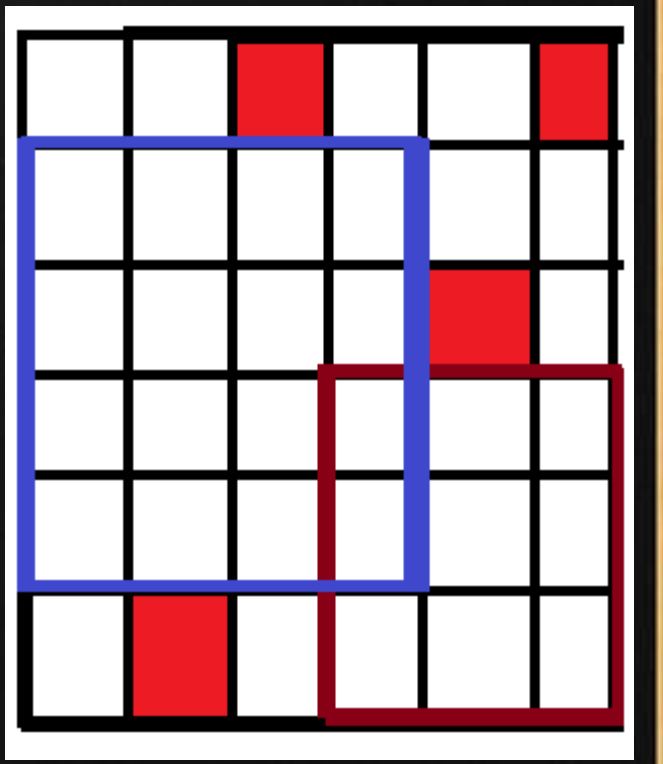
Claim: we can, for each position, try to put the first tower at its max extend (i.e. F(x, y)), and put the second tower, and we will not miss the optimal solution

i.e. DP(x, y) = F(x, y)^2 +
max(L(y - F(x,y) - 1), T(x - F(x, y) - 1)) ^ 2, given that the second term is not 0

Reason: Let's assume, the optimal solution is extend some (x, y) to some subdegree other then F(x,y)

DP problems

For example, if we extend (6, 6) to 3, we will miss the blue square, which is optimal for (6,6). It is obvious that we should fix brown to 1-degree



DP problems

But, we can always find another square in brown, so that the blue square do not touch with the max extent of brown, and it is the original size that (6,6) should extend. It's indicated by green arrows.

(Question: What does it imply if such square does not exist?)