動的計画法を極める！
蟻本3-4 + α

大規模知識処理研究室　M2
竹内文登

イントロ

- DP（動的計画法）は、
  - 分割統治法と
  - メモ化 をする探索手法。（おさらい）
- 今日は、DPのための頻出テクニックを紹介
  - おもに高速化テク、実装テク

今日の内容

- ★ 動的計画法を極める！ ← 蟻本2版 3-4
  - ★ ビットDP
  - ★ 行列累乗
  - ★ データ構造を用いて高速化

- ★ 区間DP

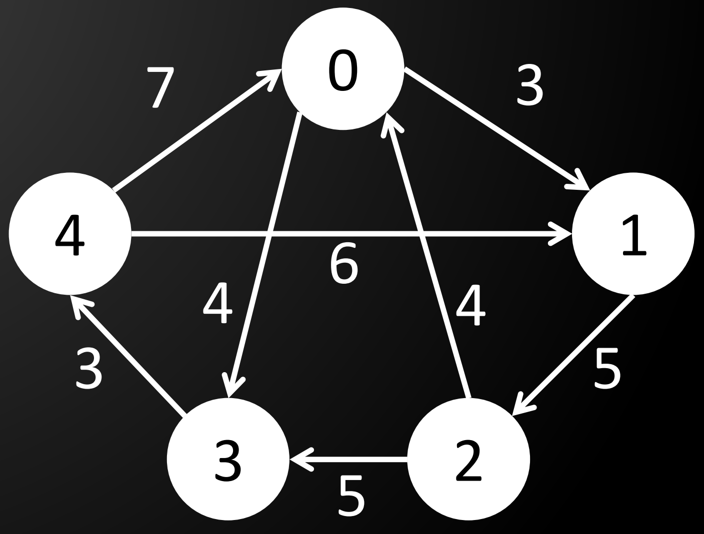
ビットDP

- 巡回セールスマン問題
- ドミノ敷き詰め問題

巡回セールスマン問題

- ★ 頂点数 n の重み付き有向グラフ
- ★ 頂点0からスタートして、すべての頂点を1度ずつめぐって帰って来る閉路のうち、
- ★ 辺の重み総和の最小値を求めよ

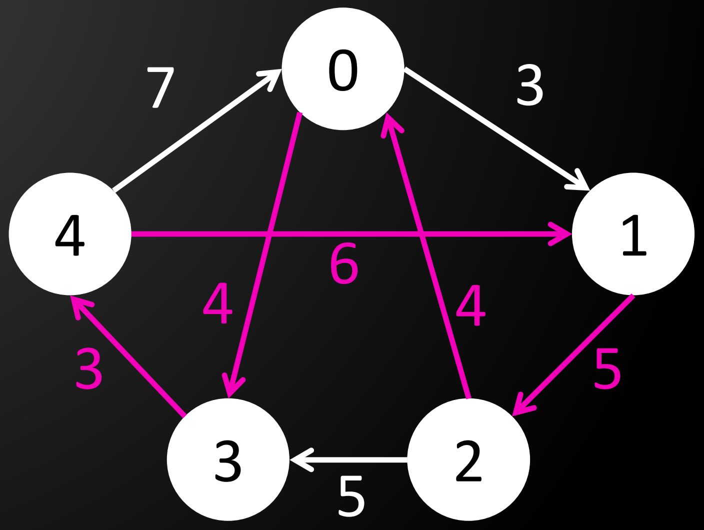
- ★ 制約
  - ★ $2 \leqq n \leqq 15$
  - ★ $0 \leqq w_i \leqq 1000$（辺の重み）



巡回セールスマン問題

- ★ 頂点数 n の重み付き有向グラフ
- ★ 頂点0からスタートして、すべての頂点を1度ずつめぐって帰って来る閉路のうち、
- ★ 辺の重み総和の最小値を求めよ

- ★ 制約
  - ★ $2 \leqq n \leqq 15$
  - ★ $0 \leqq w_i \leqq 1000$（辺の重み）



巡回セールスマン問題について

- ★ 英語で、Traveling Salesman Problem（TSP）
- ★ “NP困難”という難しいクラスの問題
  - ★ 多項式時間で解くアルゴリズムは知られていない
  - ★ つまり、小さい n に対してしか解けない問題

愚直な解法 for TSP

- ★ n 個の頂点の訪問順は $(n-1)!$ 通り
  - ★ 始点、終点は頂点0
  - ★ 残りの訪問順が $(n-1)!$ 通り

- ★ $(n-1)!$ 通りの全ての順列を計算して最小を求める

- ★ n の最大値は15（$14! = 8.7 \times 10^{10}$）
- ★ ミトメラレナイワァ

DP for TSP

- 考察
  - 頂点集合 $S \subseteq V$の頂点は辿り
  - いま $v \in S$にいるとする
  - 残りの頂点の辿り方は、S の辿り方に関係ない

  - Sと vが同じ状態について、最適な値のみ記憶
    - 辺重みの和が最小となるもの

$$\overbrace{\quad\quad\quad\quad}^{S \quad v}$$

0　1　2　3　・・・

0　2　1　3　・・・

DP for TSP

- dp[S][v] : 頂点集合 S を辿り、最後に頂点 v に至るまでの、辺重みの最小値
  - $S \subseteq V$ の取り方は 2^V 通り
  - $v \in V$ の取り方は V 通り

- 初期化 : dp[ϕ][0] = 0
- 答え : dp[V][0] : 全ての頂点を辿り、最後は0
- 漸化式
  - $dp[S][v] = \min_{k \in S-\{v\}}(dp[S-\{v\}][k] + d(k,v))$

コードの前に

- ★ 集合 S の管理をどうする？
  - ★ 効率よく扱うため、ビット列で管理 ← ビットDP

$$S = \{1, 2, 4, 6\} \Leftrightarrow 1\,0\,1\,0\,1\,1\,0$$

頂点6 ・・・ 頂点0

C++ でのビット列の扱い

- ★ 整数型で扱う
- ★ 演算
  - ★ 左シフト演算 : <<
  - ★ OR演算 : |
  - ★ AND演算 : &
  - ★ XOR演算 : ^
  - ★ 否定演算 : ~

- ★ int S;

  - ★ 1 << 5 (= 100000 = 2^5)
  - ★ 6 | 3 (= 110 | 011 = 111)
  - ★ 6 & 3 (= 110 & 011 = 010)
  - ★ 6 ^ 3 (= 110 ^ 011 = 101)
  - ★ ~6 (= 11…001 = −7)

などを駆使してビット列を扱います。
(演算の優先順位がややこしいのでカッコを多めに！)

余談

★ バグりました。

```
int main(){

	int S = 4,T = 4;
	if( S & T == 4 )
		cout << "ハラショー!!" << endl;
	else
		cout << "ｲﾐﾜｶﾝﾅｲ..." << endl;
}
```

演算子の優先順位参考URL：
http://www9.plala.or.jp/sgwr-t/c/sec14.html

余談

★ バグりました。

優先度 低

優先度 高

```
int main(){

	int S = 4,T = 4;
	if( S & T == 4 )
		cout << "ハラショー!!" << endl;
	else
		cout << "ｲﾐﾜｶﾝﾅｲ..." << endl;
}
```

演算子の優先順位参考URL:
http://www9.plala.or.jp/sgwr-t/c/sec14.html

コード for TSP

$2^V \times V$サイズの配列確保

INFで初期化

```
ll dp[ (1<<V) ][ V ];
for(int i=0; i<(1<<V); i++) for(int j=0; j<V; j++) dp[i][j] = INT_MAX;
dp[ 0 ][ 0 ] = 0;

for(int S=1; S<(1<<V); S++)
      for(int v=0; v<V; v++)
            if( contain(S,v) )
                  for(int j=0; j<V; j++)
                        dp[S][v] = min(dp[S][v], dp[S-(1<<v)][j] + G[j][v]);
cout << dp[(1<<V)-1][0] << endl;
```

コード for TSP

S = ϕ, v = 0

全てのS, vについて計算

```
ll dp[ (1<<V) ][ V ];
for(int i=0; i<(1<<V); i++) for(int j=0; j<V; j++) dp[i][j] = INT_MAX;
dp[ 0 ][ 0 ] = 0;

for(int S=1; S<(1<<V); S++)
    for(int v=0; v<V; v++)
        if( contain(S,v) )
            for(int j=0; j<V; j++)
                dp[S][v] = min(dp[S][v], dp[S-(1<<v)][j] + G[j][v]);
cout << dp[(1<<V)-1][0] << endl;
```

頂点 j から頂点 vへの辺の重み

11･･･11

dp[S][v] の値は、S-{v}のどこかの頂点からvへ移動するときのいずれか

計算量 for TSP

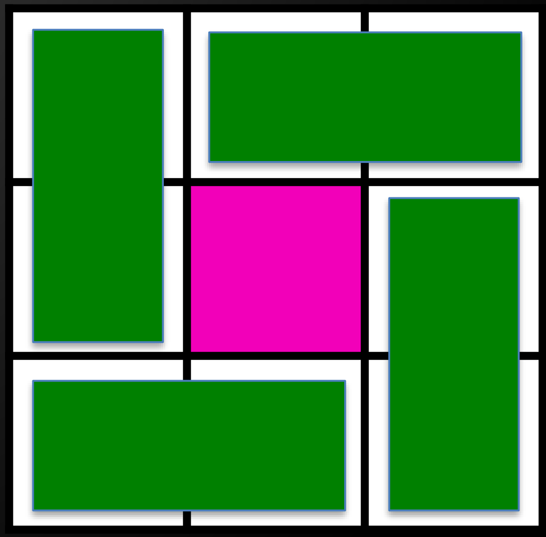
- O($2^V V^2$)

```
ll dp[ (1<<V) ][ V ];
for(int i=0; i<(1<<V); i++) for(int j=0; j<V; j++) dp[i][j] = INT_MAX;
dp[ 0 ][ 0 ] = 0;

for(int S=1; S<(1<<V); S++)
      for(int v=0; v<V; v++)
            if( contain(S,v) )
                  for(int j=0; j<V; j++)
                        dp[S][v] = min(dp[S][v], dp[S-(1<<v)][j] + G[j][v]);
cout << dp[(1<<V)-1][0] << endl;
```

ドミノ敷き詰め

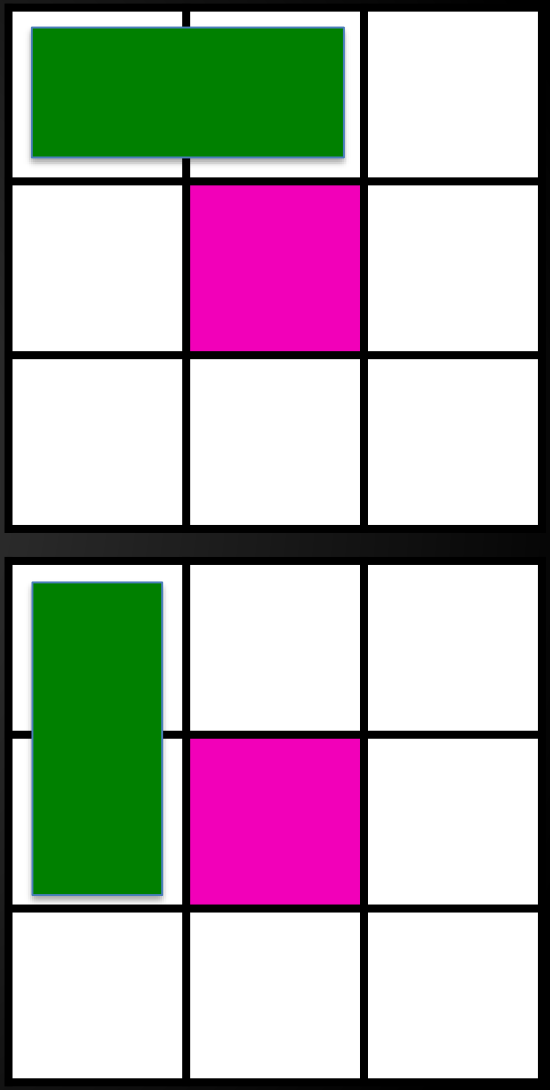
- ★ $n \times m$ のマスがある
- ★ 各マスは白か黒で塗られている
- ★ 1×2のブロックを重なりが生じないように敷き詰める
- ★ すべての白マスを覆い、黒マスを一切覆わない敷き詰め方は何通りあるか？
- ★ Mで割った余りを求めよ

- ★ $1 \leqq n,m \leqq 15$
- ★ $2 \leqq M \leqq 10^9$



愚直な解法 for ドミノ敷き詰め

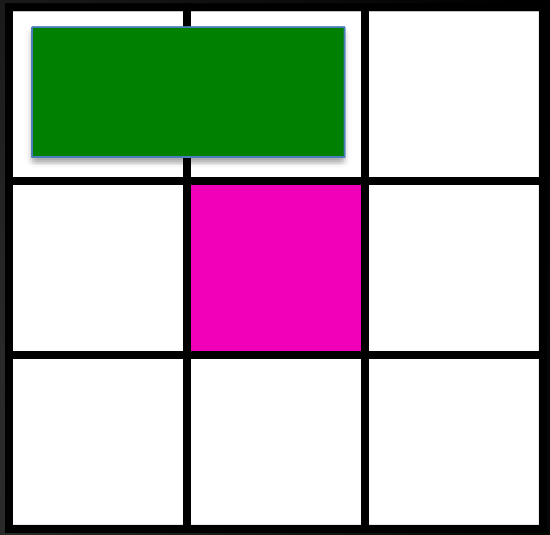
- ★ 左上のマスから順に
  - ★ よこ置き
  - ★ たて置き
- ★ で置いていく。
- ★ 最後まで置けたら ans++

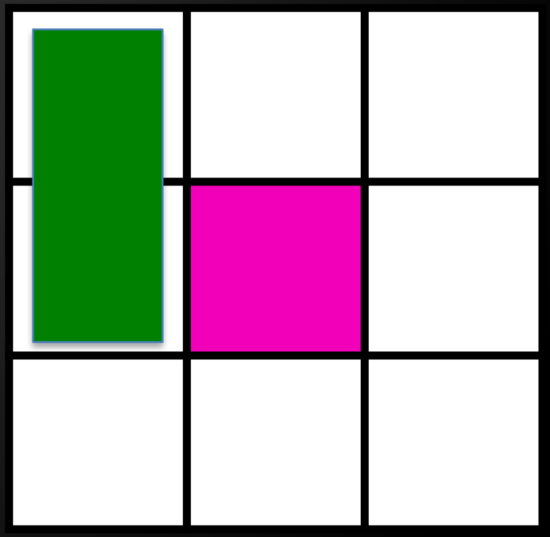


愚直な解法 for ドミノ敷き詰め

- ★ 置く場所が $n \times m$ 通り
- ★ 置き方が縦横 2 通り

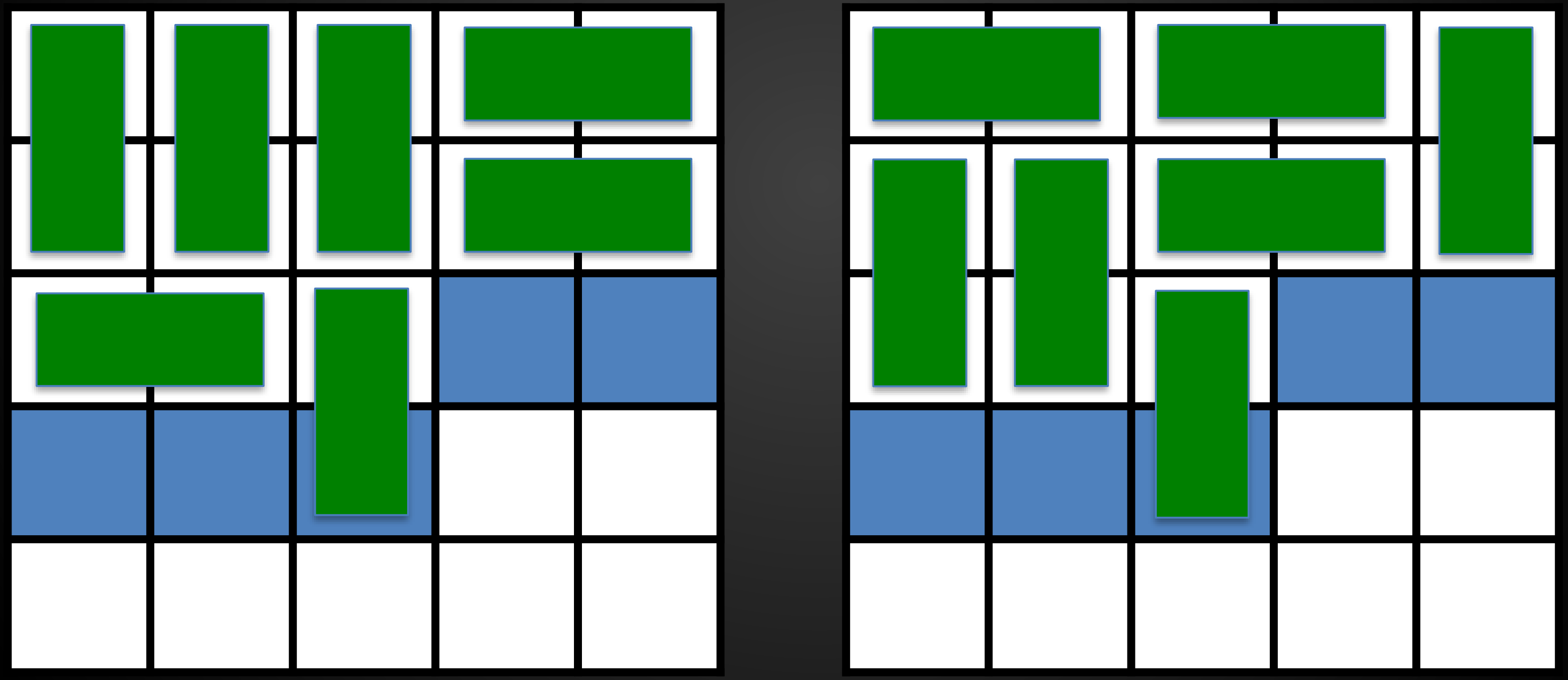
- ★ 最悪 $n \times m \times 2^{n \times m}$
  - ★ $n \times m \leqq 255$ なので無理そう





DP for ドミノ敷き詰め

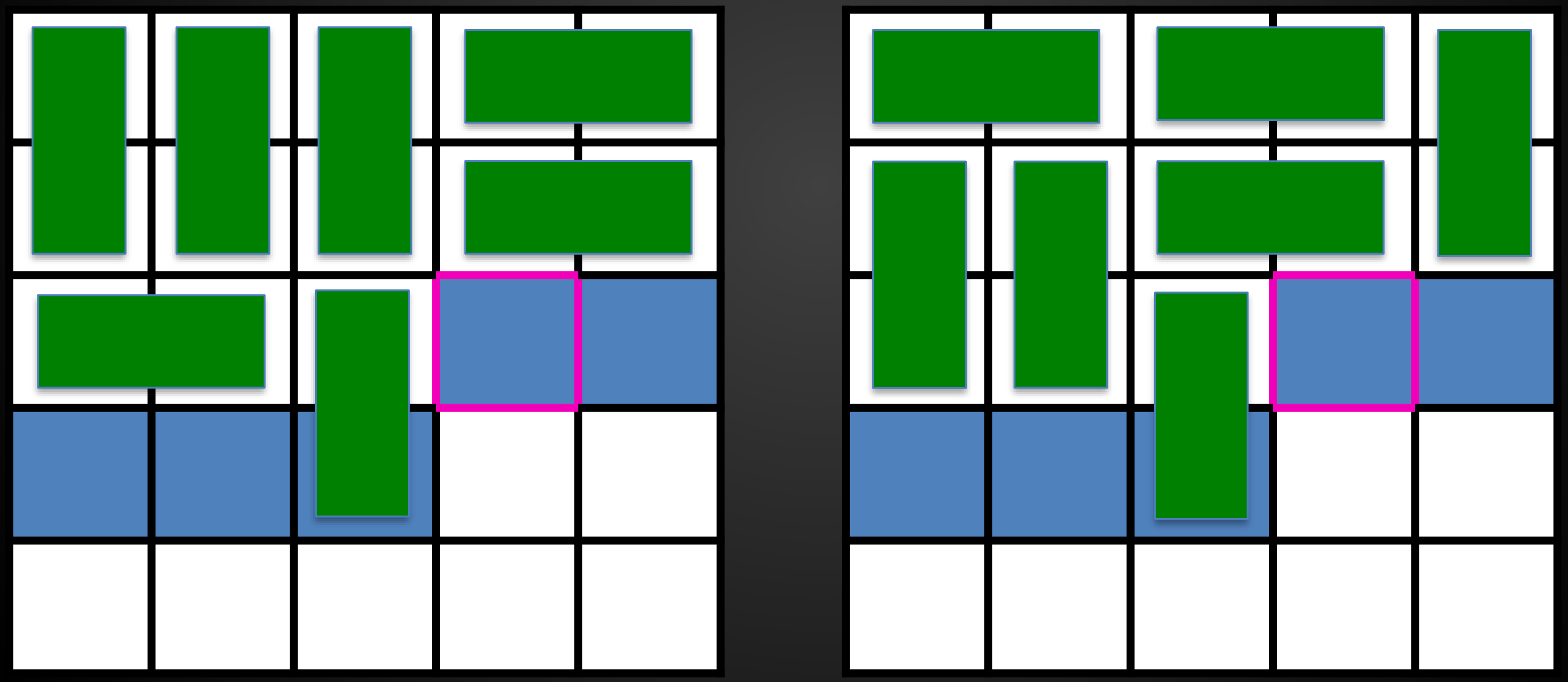
- ★ アイディア
  - ★ ギリギリの境界だけ見ればよくない？



境界が同じならそのあとの詰め方は同じ!

DP for ドミノ敷き詰め

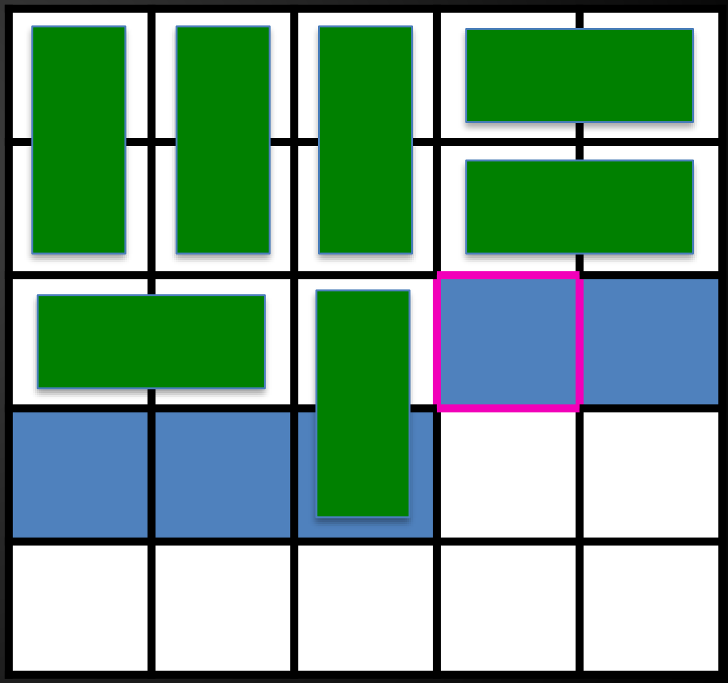
- ★ 境界をビット列で管理 ← ビットDP
  - ★ 例：00100



境界が同じならそのあとの詰め方は同じ!

DP for ドミノ敷き詰め

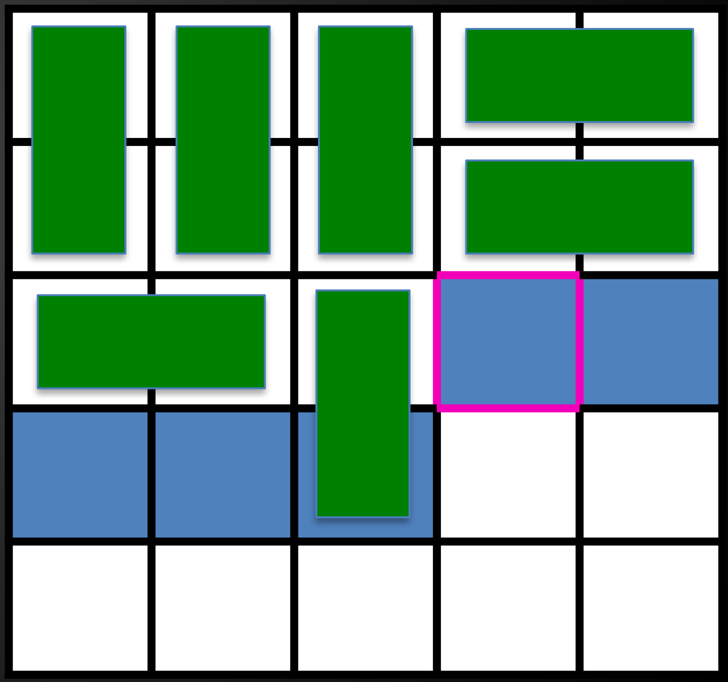
- ★ dp[i][j][S] :
  - ★ i 行 j 列まで埋めて、境界が S となるパターン数



dp[2][3][00100]に対応する状態

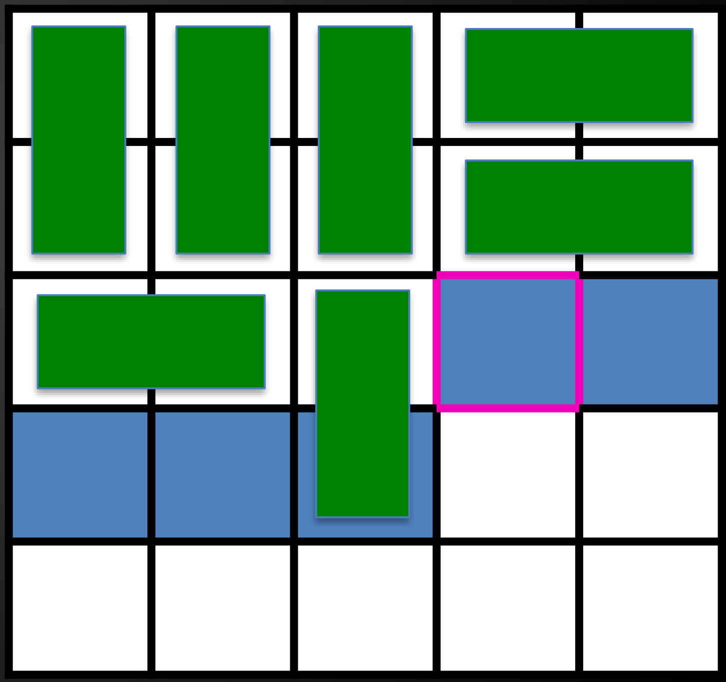
DP for ドミノ敷き詰め

- dp[i][j][S] :
  - i 行 j 列まで埋めて、境界が S となるパターン数
- 漸化式
  - (i , j) が空
    - 縦置き
    - 横置き
  - (i , j) が空でない



DP for ドミノ敷き詰め

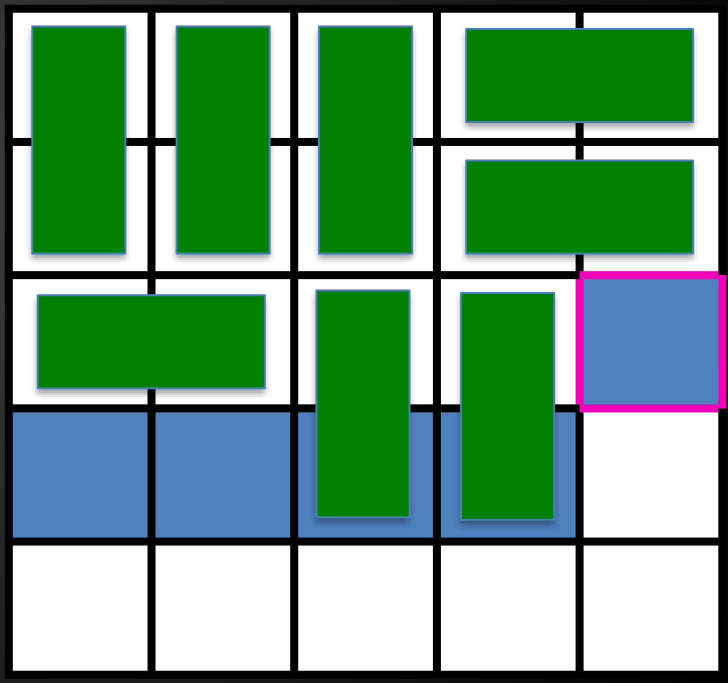
- ★ dp[i][j][S] :
  - ★ i 行 j 列まで埋めて、境界が S となるパターン数
- ★ 漸化式
  - ★ (i , j) が空
    - ★<u>縦置き</u>
    - ★横置き
  - ★ (i , j) が空でない



dp[2][3][00100]

DP for ドミノ敷き詰め

- ★ dp[i][j][S] :
  - ★ i 行 j 列まで埋めて、境界が S となるパターン数
- ★ 漸化式
  - ★（ i , j ）が空
    - ★縦置き
    - ★横置き
  - ★（ i , j ）が空でない



dp[i][j+1][S’] += dp[i][j][S]

★ S’ = S | (1<< j)

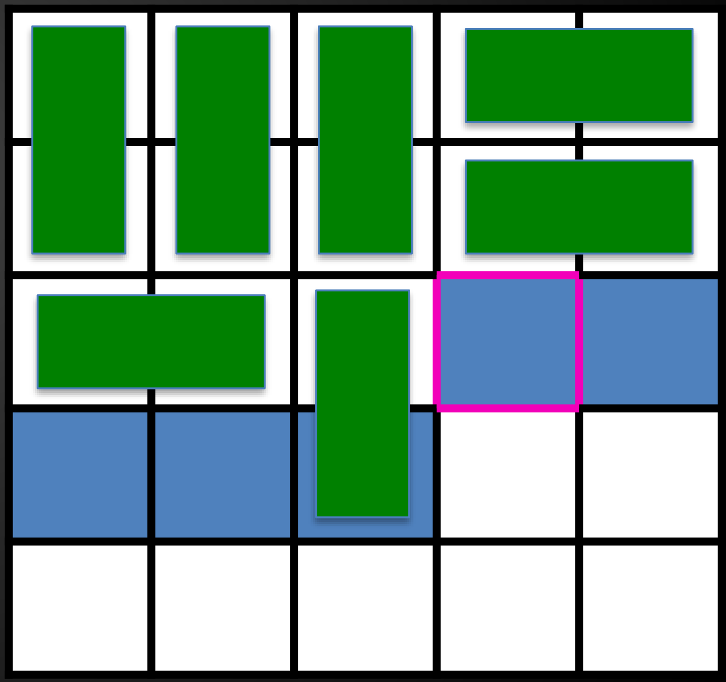
（左から j 番目の境界は埋まっている）

dp[2][3][00100]

dp[2][4][00110]

DP for ドミノ敷き詰め

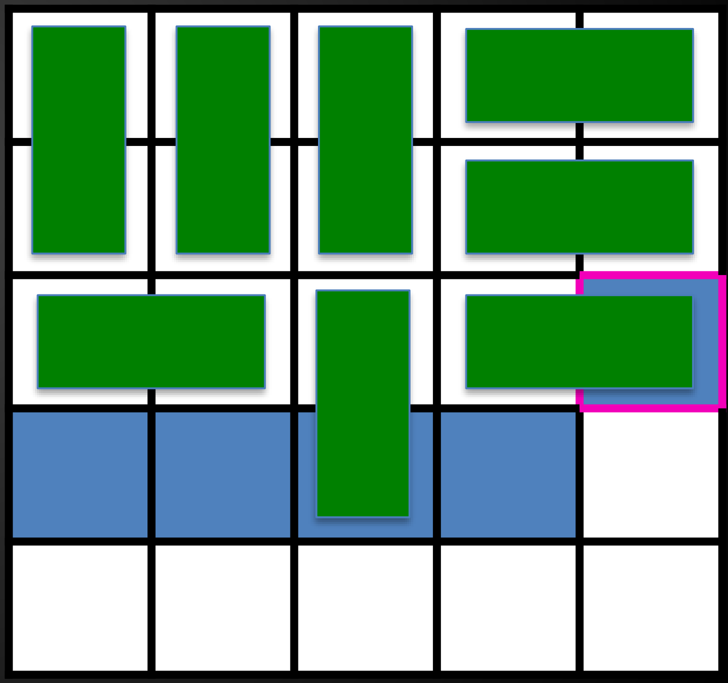
- dp[i][j][S] :
  - i 行 j 列まで埋めて、境界が S となるパターン数
- 漸化式
  - (i , j) が空
    - 縦置き
    - <u>横置き</u>
  - (i , j) が空でない



dp[2][3][00100]

DP for ドミノ敷き詰め

- ★ dp[i][j][S] :
  - ★ i 行 j 列まで埋めて、境界が S となるパターン数
- ★ 漸化式
  - ★ (i , j) が空
    - ★縦置き
    - ★<u>横置き</u>
  - ★ (i , j) が空でない



dp[i][j+1][S’] += dp[i][j][S]

★ S’ = S | (1<< (j+1))

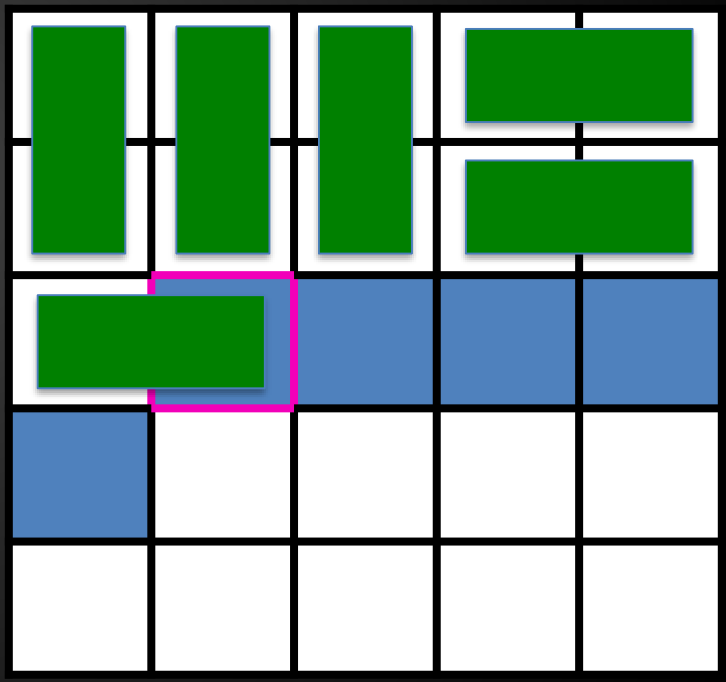
(左から j+1 番目の境界は埋まっている)

dp[2][3][00100]

dp[2][4][00101]

DP for ドミノ敷き詰め

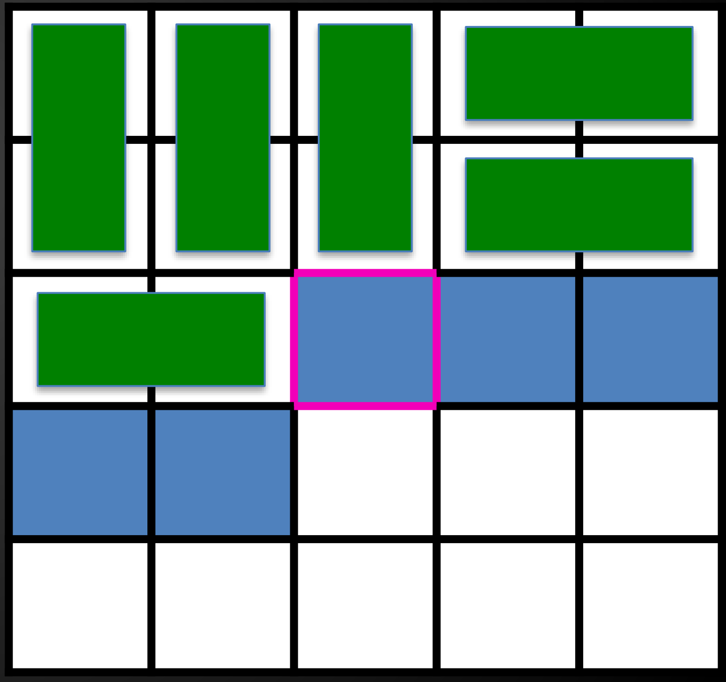
- dp[i][j][S] :
  - i 行 j 列まで埋めて、境界が S となるパターン数
- 漸化式
  - (i , j) が空
    - 縦置き
    - 横置き
  - <u>(i , j) が空でない</u>



dp[2][1][01000]

DP for ドミノ敷き詰め

- ★ dp[i][j][S] :
  - ★ i 行 j 列まで埋めて、境界が S となるパターン数
- ★ 漸化式
  - ★ (i , j) が空
    - ★縦置き
    - ★横置き
  - ★ <u>(i , j) が空でない</u>



dp[i][j+1][S'] += dp[i][j][S]

★ S' = S & ~(1<< j))

(左から j 番目の境界は空)

dp[2][1][01000]

dp[2][2][00000]

```
void solve(){
      vvvi dp(n+1,vvi(m+1,vi(1<<m,0))) ;
      dp[0][0][0] = 1;
      for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                  for(int used=0; used<(1<<m); used++){
                        if( (used & (1<<j) ) or color[i][j] ){
                              int next = used & ~(1<<j);
                              if( j+1 < m )
(i,j)が空でない                      dp[i][j+1][next] += dp[i][j][used];
                              else
                                    dp[i+1][0][next] += dp[i][j][used];
                        }
                        else{
                              if( j+1 < m and !color[i][j+1] and !(used & (1<<(j+1))) ) {
                                    int next = used | (1<<(j+1));
                   横置き             dp[i][j+1][next] += dp[i][j][used];
                              }
                              if( i+1 < n and !color[i+1][j] ) {
                                    int next = used | (1<<j);
(i,j)が空                           if( j+1 < m )
                                          dp[i][j+1][next] += dp[i][j][used];
                   縦置き             else
                                          dp[i+1][0][next] += dp[i][j][used];
                              }
                        }
                  }
            }
      }
      cout << dp[n-1][m-1][(1<<m-1)] << endl;
}
```

計算量

- ★ ループの回数から、$n \times m \times 2^m$
  - ★ $15 \times 15 \times 2^{15} = 7372800$

発展（メモリ節約）

- dp[i][j][*] の計算に必要な情報は、
- dp[i][j-1][*]のいずれかにある。

- 各 (i , j) でdp[i][j][*] を覚える必要はなく
- 一つ前のセルの dp[i][j-1][*] だけでよい。

- dp[S] を二つ使うだけでよい。
  - corr[S]
  - next[S]

発展（メモリ節約）

```
void solve(){
	vi corr(1<<MAX_M,0);
	vi next(1<<MAX_M,0);
	corr[0] = 1;
	for(int i=n-1; i>= 0; i--) {
		for(int j=m-1; j>=0; j--) {
			for(int used=0; used<(1<<m); used++){
				if(( used & (1<<j) ) or color[i][j]) {
					next[used] = corr[used & ~(1<<j)];
				}
				else {
					int res = 0;
					if( j+1 < m and !( used & (1<<(j+1) )) and !color[i][j+1] ){
						res += corr[used | 1<<(j+1)];
					}
					if( i+1 < n and !color[i+1][j] ){
						res += corr[used | 1<<j];
					}
					next[used] = res % M;
				}
			}
			swap(corr,next);
		}
	}
	cout << corr[0] << endl;
}
```

2^V配列を2つ用意

次のセルを見るタイミングで
交換して再利用

今日の内容

- ★ 動的計画法を極める！ ← 蟻本2版 3-4
  - ~~★ ビットDP~~
  - ★ 行列累乗
  - ★ データ構造を用いて高速化

- ★ 区間DP

行列累乗

- ★ フィボナッチ数列
- ★ Blocks
- ★ グラフの長さ k のパスの総数

フィボナッチ数列

- ★ フィボナッチ数列の第 n 項を求めよ
  - ★ $F_0 = 0$
  - ★ $F_1 = 1$
  - ★ $F_{n+2} = F_{n+1} + F_n$ （$2 \leqq n$）
- ★ 制約
  - ★ $0 \leqq n \leqq 10^{16}$

愚直な解法 for Fibonacci Number

- 漸化式や！DP！DP！
  - F_0 から F_n まで順に計算していく。

- *え？ n でかくね…*
  - $n \leqq 10^9$ までならいいが…

行列累乗

★ フィボナッチ数列の漸化式を行列で書くと…

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n+1} \\ F_{n} \end{bmatrix}$$

★ 再帰的に以下のように書ける。

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n} \begin{bmatrix} F_{1} \\ F_{0} \end{bmatrix}$$

↑

“繰り返し二乗法” で高速に計算可能！

コード for Fibonacci Number

```
ll Fib(ll n){
      mat A = {{1,1},{1,0}};
      mat An = mat_pow(A,n);

      mat b = {{1},{0}};
      mat ans = mat_times(An,b);

      return ans[1][0];
}
```

繰り返二乗法
O(log n)

行列積

コード for Fibonacci Number

参考：繰り返二乗法 O(log n)

```
mat mat_pow(mat A, ll n){
	if( n==0 ){
		mat E = {{1,0},{0,1}};
		return E;
	}
	if( n%2 == 0 )
		return mat_pow(mat_times(A,A),n/2);
	else
		return mat_times(mat_pow(mat_times(A,A),n/2),A) ;
}
```

m 項間漸化式の場合

★ 漸化式: $a_{n+m} = \sum_{i=0}^{m-1} b_i a_{n+i}$

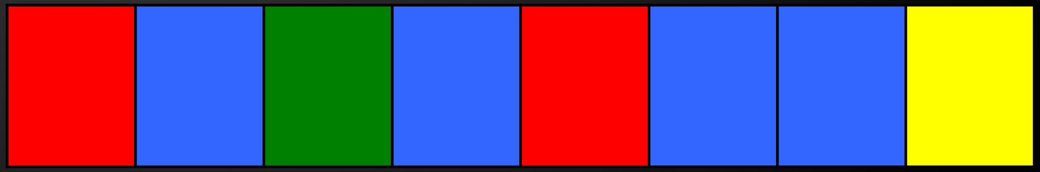
$$\begin{bmatrix} a_{n+m} \\ a_{n+m-1} \\ \dots \\ a_{n+1} \end{bmatrix} = \begin{bmatrix} b_{m-1} & \dots & b_1 & b_0 \\ 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n+m-1} \\ a_{n+m-2} \\ \dots \\ a_n \end{bmatrix}$$

この係数行列を *n* 乗する。$O(m^3 \log n)$

Blocks (POJ 3734)

- ★ N 個のブロックが一列に並んでいる
- ★ 各ブロックを、赤、青、緑、黄のいずれかで塗る
- ★ 赤、緑で塗られたブロックの個数が、ともに偶数となる塗り方は何通りあるか？
- ★ 10007で割った余りを求めよ

- ★ 制約
  - ★ $1 \leqq N \leqq 10^9$



愚直?な解法 for Blocks

- 前から何を塗るかで場合分け
  - 各ブロックは4色塗れるので $O(4^N)$ で計算可(むり)

愚直?な解法 for Blocks

- 前から何を塗るかで場合分け
  - 各ブロックは4色塗れるので $O(4^N)$ で計算可（むり）
- DPじゃね？
  - 赤と緑の偶奇の情報のみ覚えればよい
  - 前から計算で O(N)でいけそう

愚直?な解法 for Blocks

- 前から何を塗るかで場合分け
  - 各ブロックは4色塗れるので $O(4^N)$ で計算可(むり)
- DPじゃね？
  - 赤と緑の偶奇の情報のみ覚えればよい
  - 前から計算で O(N)でいけそう
- 制約が $N \leqq 10^9$ なので O(N) の解法は厳しい

DP for Blocks

- ★ a_i : i 個目まで赤緑ともに偶数個となる塗り方の数
- ★ b_i : i 個目まで赤緑一方が奇数個となる塗り方の数
- ★ c_i : i 個目まで赤緑ともに奇数個となる塗り方の数

DP for Blocks

- a_i : i 個目まで赤緑ともに偶数個となる塗り方の数
- b_i : i 個目まで赤緑一方が奇数個となる塗り方の数
- c_i : i 個目まで赤緑ともに奇数個となる塗り方の数

- 漸化式

$$a_{i+1} = 2 * a_i + b_i$$

$$b_{i+1} = 2 * a_i + 2 * b_i + 2 * c_i$$

$$c_{i+1} = b_i + 2 * c_i$$

DP for Blocks

★ 漸化式を行列変換 → 繰り返し二乗法で高速化

$$a_{i+1} = 2 * a_i + b_i$$

$$b_{i+1} = 2 * a_i + 2 * b_i + 2 * c_i$$

$$c_{i+1} = b_i + 2 * c_i$$

$$\Updownarrow$$

$$\begin{pmatrix} a_{i+1} \\ b_{i+1} \\ c_{i+1} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 2 & 2 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} a_i \\ b_i \\ c_i \end{pmatrix}$$

DP for Blocks

★ 漸化式を行列変換 → 繰り返し二乗法で高速化

$$a_{i+1} = 2 * a_i + b_i$$

$$b_{i+1} = 2 * a_i + 2 * b_i + 2 * c_i$$

$$c_{i+1} = b_i + 2 * c_i$$

$$\Updownarrow$$

$$\begin{pmatrix} a_n \\ b_n \\ c_n \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 2 & 2 \\ 0 & 1 & 2 \end{pmatrix}^n \begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix}$$

DP for Blocks

- 計算量 O(log n)
  - 繰り返し二乗法より O(log n)回の行列積
  - 3×3の行列積は $3^3 = O(1)$

$$\Updownarrow$$

$$\begin{pmatrix} a_n \\ b_n \\ c_n \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 2 & 2 \\ 0 & 1 & 2 \end{pmatrix}^n \begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix}$$

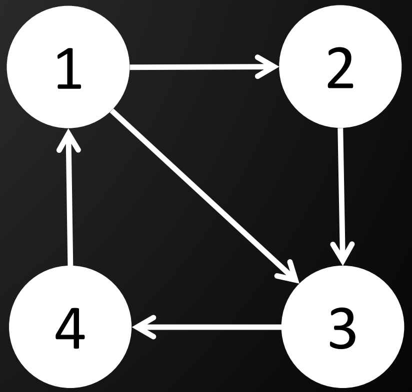
コード for Blocks

★ A を N 乗するだけ

```
int main(){

	ll N; cin >> N;

	mat A = {{2,1,0},{2,2,2},{0,1,2}};
	mat a0 = {{1},{0},{0}};

	A = mat_pow(A,N);
	mat ans = mat_prod(A,a0);

	cout << ans[0][0] << endl;
}
```

グラフの長さ k のパスの総数

- ★ 頂点数 n のグラフの隣接行列が与えられる
- ★ すべての辺の長さは1
- ★ このグラフ中の長さ k のパスの総数はいくつか
  - ★ 同じ辺を何度通るパスも含む
- ★ 10007で割った余りを求めよ



- ★ 制約
  - ★ $1 \leqq n \leqq 100$
  - ★ $1 \leqq k \leqq 10^9$

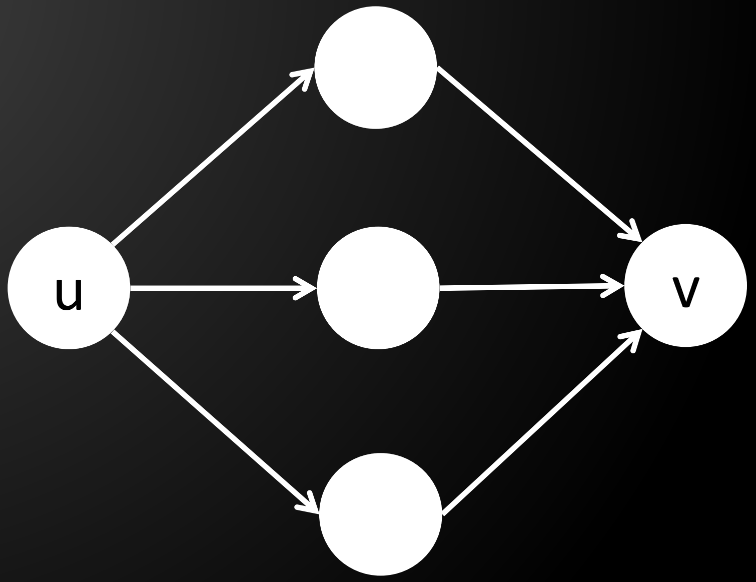
グラフの長さ k のパスの総数 –考察–

- ★ G_k[u][v] : u から vへの長さ k パスの総数

$$G_{k_1+k_2}[u][v] = \sum_{w=1}^{n} G_{k_1}[u][w] * G_{k_2}[w][v]$$

- ★ が成立するため

$$G_{k_1+k_2} = G_{k_1} * G_{k_2}$$



- ★ と行列積で計算できる
- ★ G_1 隣接行列そのもの

グラフの長さ k のパスの総数

- ★ つまり、
- ★ 隣接行列 G を k 乗すると、
- ★ G[u][v]は u から v へのパスの総数となる

- ★ G^kの全要素の和が答えとなる
  - ★ 繰り返し二乗法により $O(n^3 \log k)$

コード for グラフの長さ k のパスの総数

- ★ Mat G;
- ★ Mat_pow(G,k);　←　k 乗して、

ほぼ一緒なので割愛します。

- ★ ll Ans = 0;
- ★ for(int i=0; i<n; i++)
  - ★ for(int j=0; j<n; j++)
    - ★Ans += G[i][j];　←　和を計算する。
    - ★Ans %= mod;
- ★ cout << Ans << endl;

Matrix Power Series (POJ3233)

- n×n 行列 A, 正の整数 k, M が与えられる
- 以下の行列の累乗和を求め、
- 各要素をMで割った値を求めなさい

$$S = A + A^2 + \cdots A^k$$

- 制約
  - $1 \leqq n \leqq 30$
  - $1 \leqq k \leqq 10^9$
  - $1 \leqq M \leqq 10^4$

愚直な解法 for Matrix Power Series

- ★ A^k は $O(n^3 \log k)$で計算可能
- ★ すべての k に対して A^k を計算して
- ★ 全部足すことでSを求める。

- ★ $O(n^3 k \log k + n^2 k) = O(n^3 k \log k)$で無理ぽ

解法 for Matrix Power Series

★ これも繰り返し二乗法で解ける
  ★ $S_k = I + A + A^2 + \cdots A^{k-1}$ とすると

$$\left(\frac{A^k}{S_k}\right) = \left(\begin{array}{c|c} A & 0 \\ \hline I & I \end{array}\right)\left(\frac{A^{k-1}}{S_{k-1}}\right) = \left(\begin{array}{c|c} A & 0 \\ \hline I & I \end{array}\right)^k \left(\frac{I}{0}\right)$$

★ となるので、この行列を k 乗すればよい。

コード for Matrix Power Series

```
void solve(){

    mat B(n*2,vll(n*2,0));
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            B[i][j] = A[i][j];
        }
        B[n+1][i] = B[n+i][n+i] = 1;
    }
    B = mat_pow(B,k+1); // I+A+A^2+ ... +A^k
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            int a = B[n+i][j] % M;
            if( i==j ) a = (a+M-1) % M;
            cout << a;
            if(j+1==n) cout << endl;
            else cout << " ";
        }
    }
}
```

今日の内容

- ★ 動的計画法を極める！ ← 蟻本2版 3-4
  - ~~★ ビットDP~~
  - ~~★ 行列累乗~~
  - ★ データ構造を用いて高速化

- ★ 区間DP

データ構造を用いて高速化

★ Minimizing maximizer

Minimizing maximizer (POJ 1769)

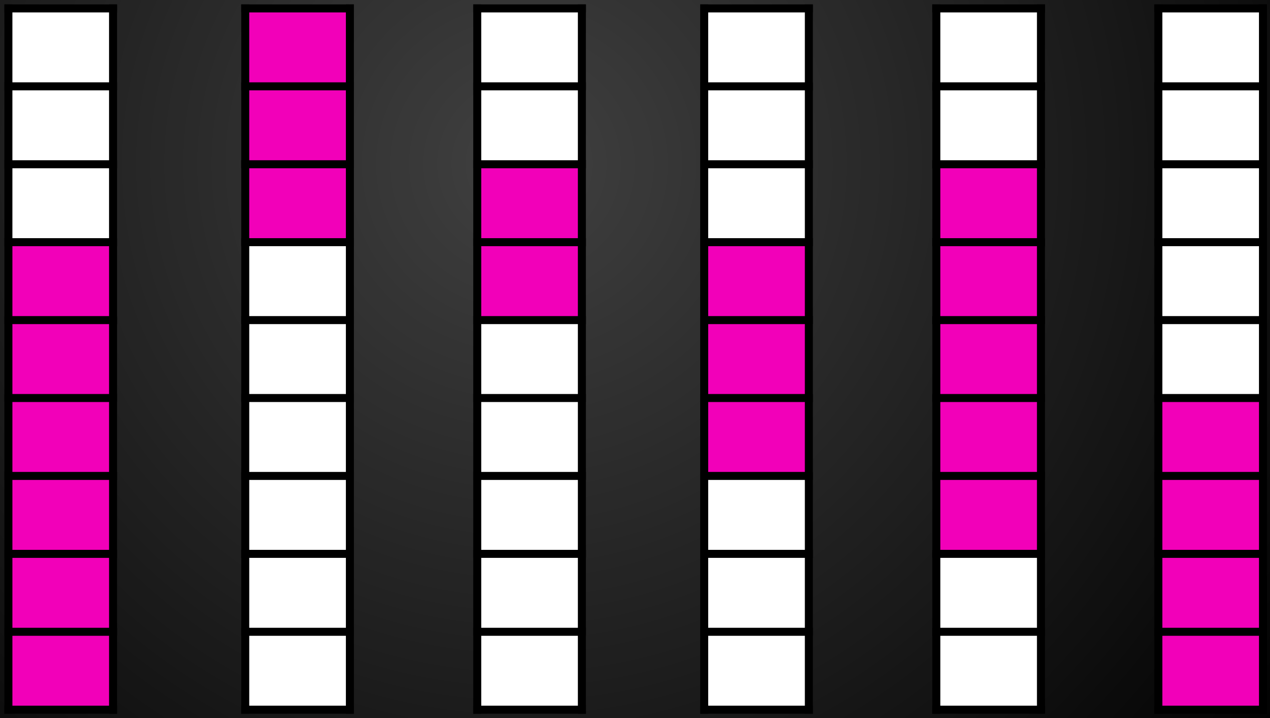
- 数列を部分的にソートする機械(Sorter)が m 個並んでいる
  - k 番目の Sorter は k−1番目の Sorterの出力を入力とする
  - k 番目の Sorter は s_k ～ t_k の区間をソートした数列を出力
  - 1番目の Sorter 入力は、与えられる数列である
  - この機械の出力は、m番目の Sorter の出力の n 番目の値
- m 個のSorterうち、いくつかの Sorter を取り除いても、常に最大値が出力が得られる場合がある。
- Sorterの列が与えられるので、常に最大値が出力されるようなSorterの最小の個数を求めよ

- 制約
  - $2 \leqq n \leqq 50000,\ 1 \leqq m \leqq 500000,\ 1 \leqq s_k < t_k \leqq n$

Minimizing maximizer 例

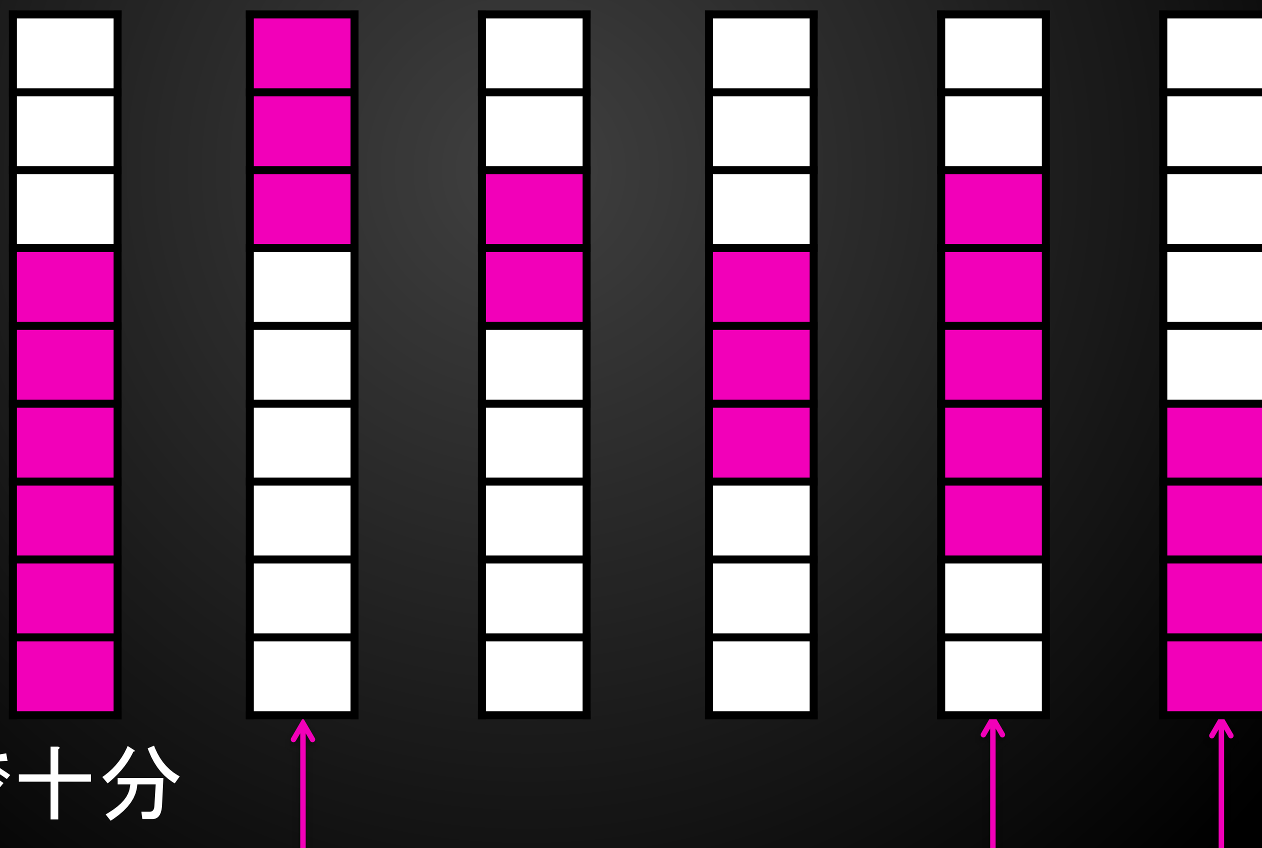
★ n = 6, m = 6

★ (s,t) = { (4,9), (1,3), (3,4), (4,6), (3,7), (6,9) }



Minimizing maximizer 例

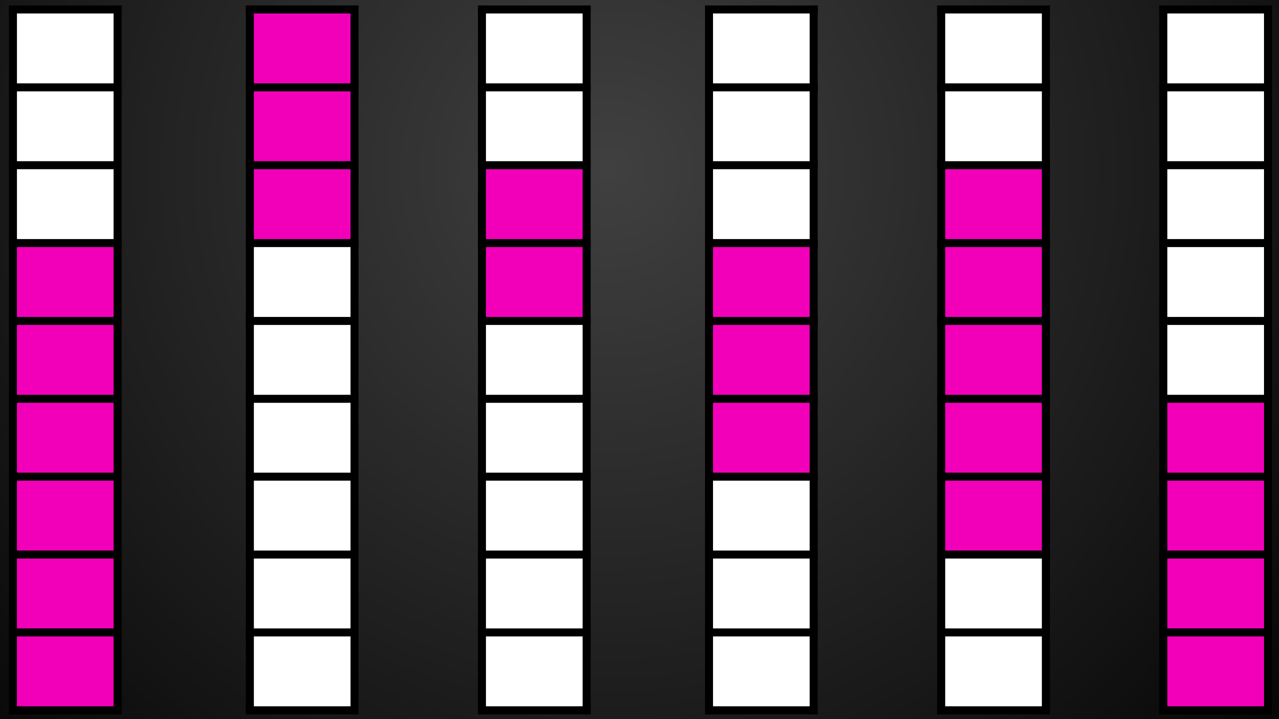
- ★ n = 6, m = 6
- ★ (s,t) = { (4,9), (1,3), (3,4), (4,6), (3,7), (6,9) }



この3つで十分

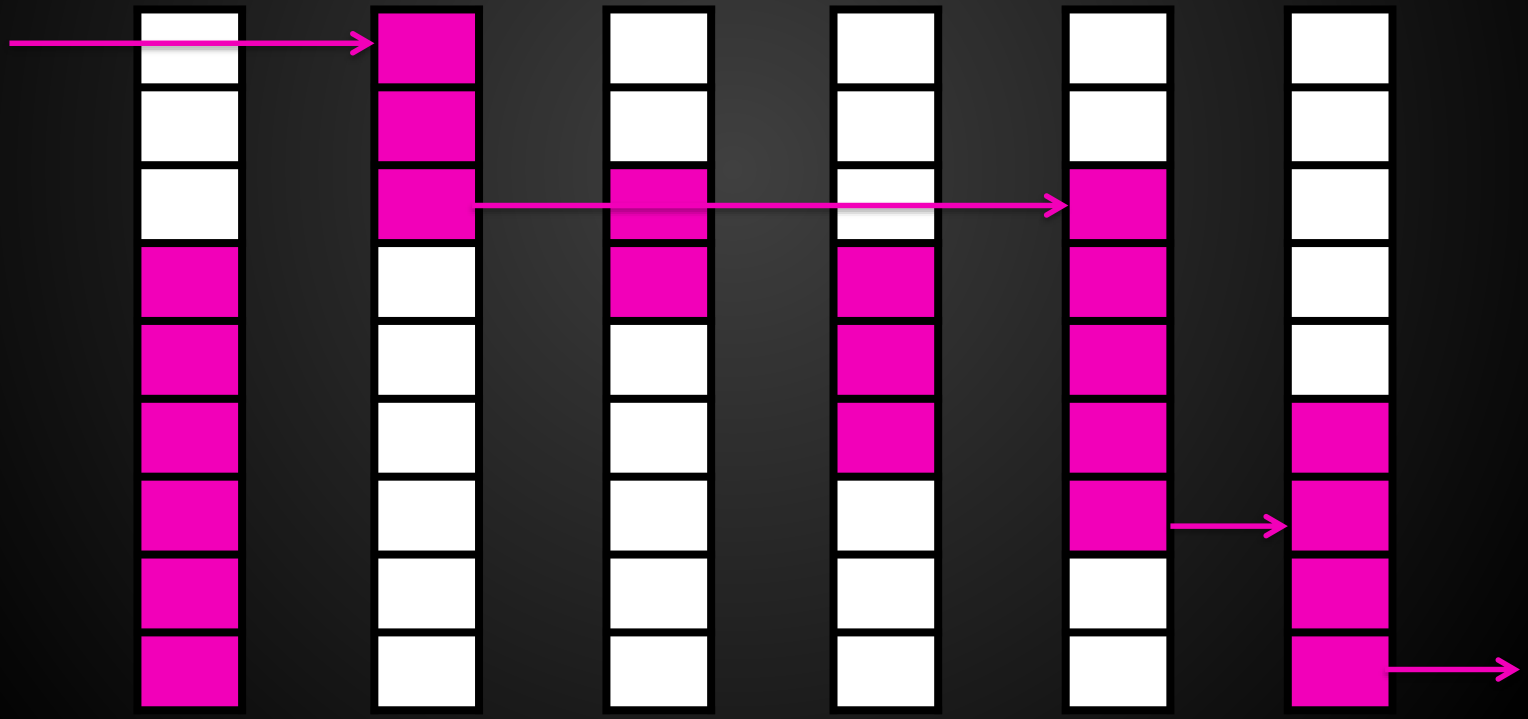
Minimizing maximizer -考察-

★ 常に最大値が得られるのはどのような場合か？



Minimizing maximizer –考察–

- 常に最大値が得られるのはどのような場合か？
  - 1番目の要素が最大値の場合に出力できるとき



DP for Minimizing maximizer

★ dp[i][j] : 1番目の要素が、i 番目までの Sorter を使って位置 j に移動するのに必要な最小の Sorter の数

$$dp[i+1][t_i] = min(dp[i][j], \min\{dp[i][j'] \mid s_i \leq j' \leq t_i\} + 1)$$

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2									
(3,4)	3									
(4,6)	4									
(3,7)	5									
(6,9)	6									

（空欄は+∞で初期化）

DP for Minimizing maximizer

★ dp[i][j] : 1番目の要素が、i 番目までの Sorter を使って位置 j に移動するのに必要な最小の Sorter の数

$$dp[i+1][t_i] = min(dp[i][t_i], \min\{dp[i][j'] \mid s_i \leq j' \leq t_i\} + 1)$$

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2	0		1						
(3,4)	3									
(4,6)	4									
(3,7)	5									
(6,9)	6									

（空欄は+∞で初期化）

DP for Minimizing maximizer

★ dp[i][j] : 1番目の要素が、i 番目までの Sorter を使って位置 j に移動するのに必要な最小の Sorter の数

$$dp[i+1][t_i] = min(dp[i][t_i], \min\{dp[i][j'] \mid s_i \leq j' \leq t_i\} + 1)$$

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2	0		1						
(3,4)	3	0		1	2					
(4,6)	4									
(3,7)	5									
(6,9)	6									

（空欄は+∞で初期化）

DP for Minimizing maximizer

★ dp[i][j] : 1番目の要素が、i 番目までの Sorter を使って位置 j に移動するのに必要な最小の Sorter の数

$$dp[i+1][t_i] = min(dp[i][t_i], \min\{dp[i][j'] \mid s_i \le j' \le t_i\}+1)$$

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2	0		1						
(3,4)	3	0		1	2					
(4,6)	4	0		1	2		3			
(3,7)	5									
(6,9)	6									

（空欄は+∞で初期化）

DP for Minimizing maximizer

★ dp[i][j] : 1番目の要素が、i 番目までの Sorter を使って位置 j に移動するのに必要な最小の Sorter の数

$$dp[i+1][t_i] = min(dp[i][t_i], \min\{dp[i][j'] \mid s_i \leq j' \leq t_i\} + 1)$$

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2	0		1						
(3,4)	3	0		1	2					
(4,6)	4	0		1	2		3			
(3,7)	5	0		1	2		3	2		
(6,9)	6									

（空欄は+∞で初期化）

DP for Minimizing maximizer

★ dp[i][j] : 1番目の要素が、i 番目までの Sorter を使って位置 j に移動するのに必要な最小の Sorter の数

$$dp[i+1][t_i] = min(dp[i][t_i], \min\{dp[i][j'] \mid s_i \leq j' \leq t_i\}+1)$$

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2	0		1						
(3,4)	3	0		1	2					
(4,6)	4	0		1	2		3			
(3,7)	5	0		1	2		3	2		
(6,9)	6	0		1	2		3	2		3

（空欄は+∞で初期化）

DP for Minimizing maximizer

★ dp[i][j] : 1番目の要素が、i 番目までの Sorter を使って位置 j に移動するのに必要な最小の Sorter の数

$$dp[i+1][t_i] = min(dp[i][t_i], \min\{dp[i][j'] \mid s_i \leq j' \leq t_i\} + 1)$$

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2	0		1						
(3,4)	3	0		1	2					
(4,6)	4	0		1	2		3			
(3,7)	5	0		1	2		3	2		
(6,9)	6	0		1	2		3	2		$\boxed{3}$

（空欄は+∞で初期化）

DP for Minimizing maximizer

- 計算量 O(nm)
  - m 回以下を繰り返す
    - 各行のすべての要素を更新：O(n)
    - 1箇所のみ区間の最小値を調べる：O(n)
- 2 ≦ n ≦ 50000, 1 ≦ m ≦ 500000 なので無理

sort	DP	1	2	3	4	5	6	7	8	9
	0	0								
(4,6)	1	0								
(1,3)	2	0		1						
(3,4)	3	0		1	2					
(4,6)	4	0		1	2		3			
(3,7)	5	0		1	2		3	2		
(6,9)	6	0		1	2		3	2		$\boxed{3}$

高速化DP for Minimizing maximizer

- 1箇所以外は dp[i+1][j] = dp[i][j] で無駄なので、1次元配列で管理する
  - m 回区間の最小値を求めるになる
- 区間の最小値を求めるのにセグメントツリーを使う
  - 区間の最小値はO(log n)で求めることができる
- つまりO(m log n)

sort	1	2	3	4	5	6	7	8	9
(4,6)	0		1	2		3	2		

コード for minimizing maximizer

```
void solve(){

    seg_init(); // Segment Tree init.
    fill(dp, dp+n+1, INT_MAX);
    dp[1] = 0;
    update(1,0);
    for(int i=0; i<m; i++){
        int v = min(dp[t[i]], query(s[i],t[i]+1, 0, 0, n) + 1);
        dp[t[i]] = v;
        update(t[i],v);
    }
    cout << dp[n_] << endl;
}
```

区間DP

- ★ 連鎖行列積
- ★ ICPC2016国内予選 D問題「ダルマ落とし」
- ★「Bribe the Prisoners」

連鎖行列積

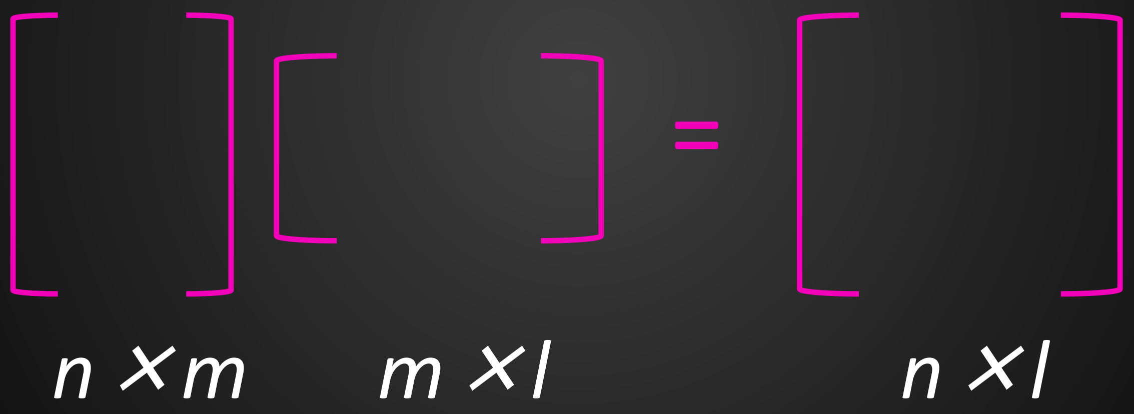
- ★ n 個の行列の連鎖がある
- ★ 各行列の次元（行数 r_i、列数 c_i）が与えられる
- ★ n 個の行列の積を計算するために必要な最小のスカラー乗算の回数を求めよ

- ★ $1 \leq n \leq 100$
- ★ $1 \leq r, c \leq 100$

$$\begin{bmatrix} \\ (30,35) \\ \\ \end{bmatrix} \begin{bmatrix} \\ (35,15) \\ \\ \end{bmatrix} \begin{bmatrix} \\ (15,5) \\ \\ \end{bmatrix} \begin{bmatrix} \\ (5,10) \\ \\ \end{bmatrix}$$

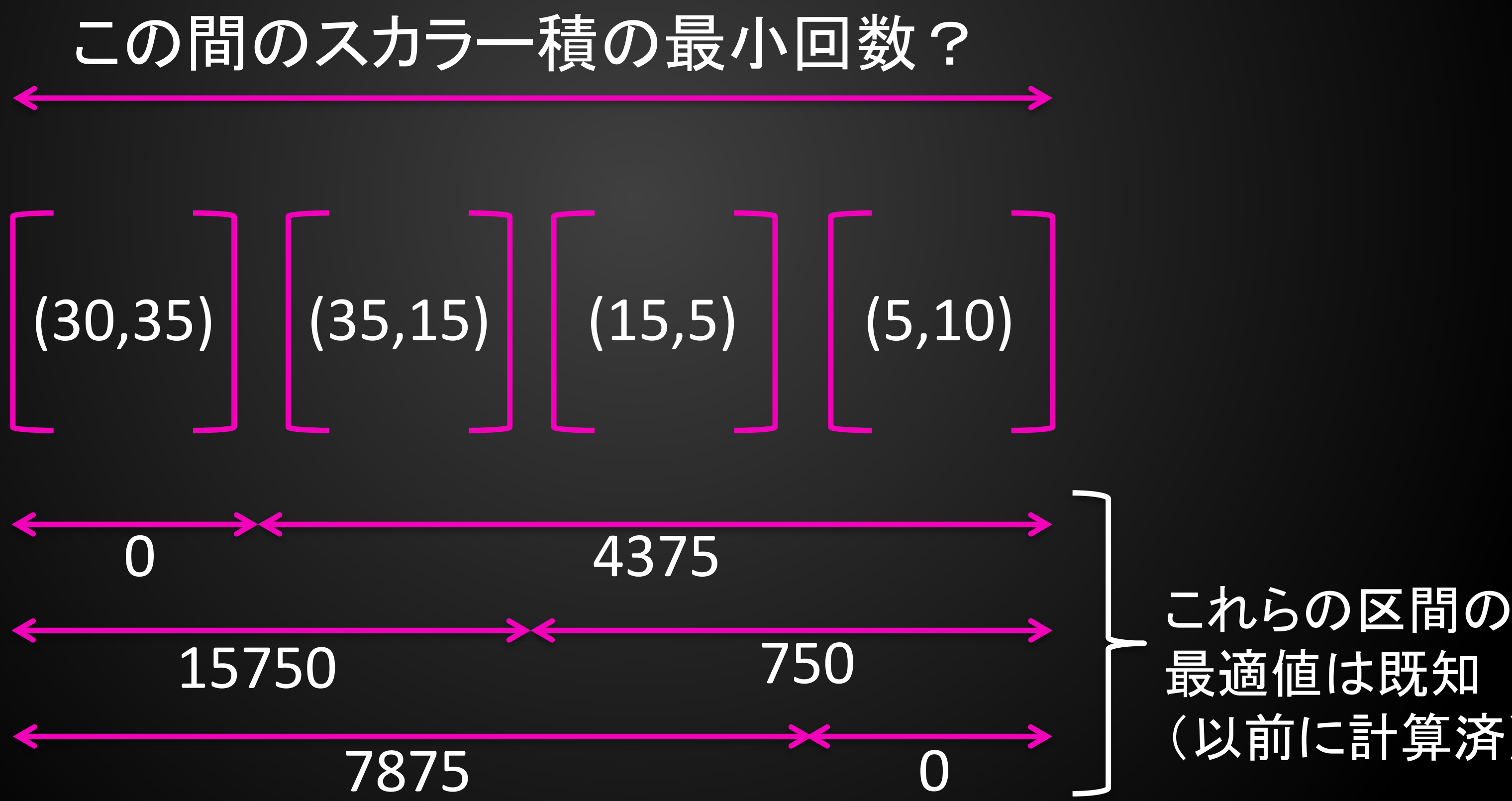
行列積（おさらい）

- $n \times m$ 行列と $m \times l$ 行列の乗算
  - $n \times m \times l$ 回のスカラー積が必要



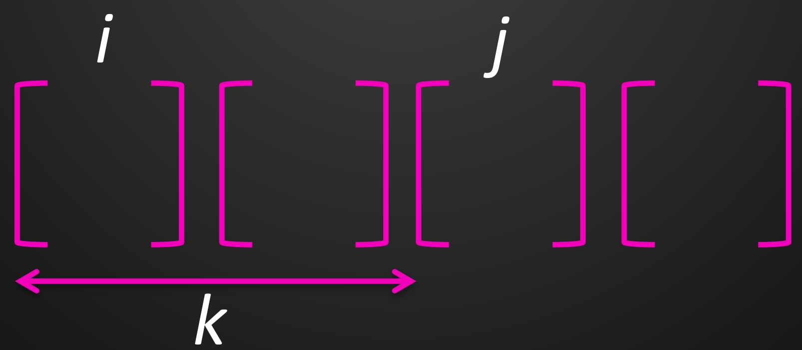
DP for 連鎖行列積

- ★ 区間DP
  - ★ 短い区間の結果を用いて、長い区間を計算



DP for 連鎖行列積

- dp[i][j] : i から j 番目の行列積に必要なスカラー積の最小値
- 漸化式
  - dp[i][j] = $\min_k$(dp[i][k] + dp[k+1][j] + prod)
  - prod = M[i].row × M[k].col × M[j].col

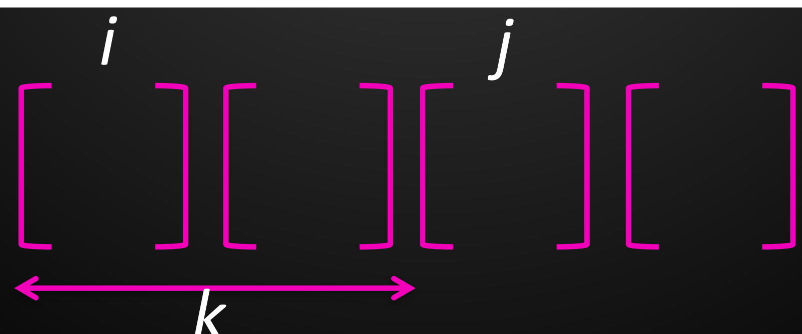


i , *j* の幅が小さい区間から計算をする

コード for 連鎖行列積

```
int dp[n][n] = {};
for(int i=0; i<n; i++)
      for(int j=0; j<n; j++)
            if(i==j) dp[i][j] = 0;
            else dp[i][j] = INT_MAX;

for(int w=1; w<n; w++){
      for(int i=0; i+w<n; i++){
            int j = i+w;
            for(int k=i; k<j; k++){
                  int num_prod = M[i].r * M[k].c * M[j].c;
                  dp[i][i+w] = min(dp[i][i+w], dp[i][k] + dp[k+1][i+w] + num_prod);
            }
      }
}
cout << dp[0][n-1] << endl;
```

初期化



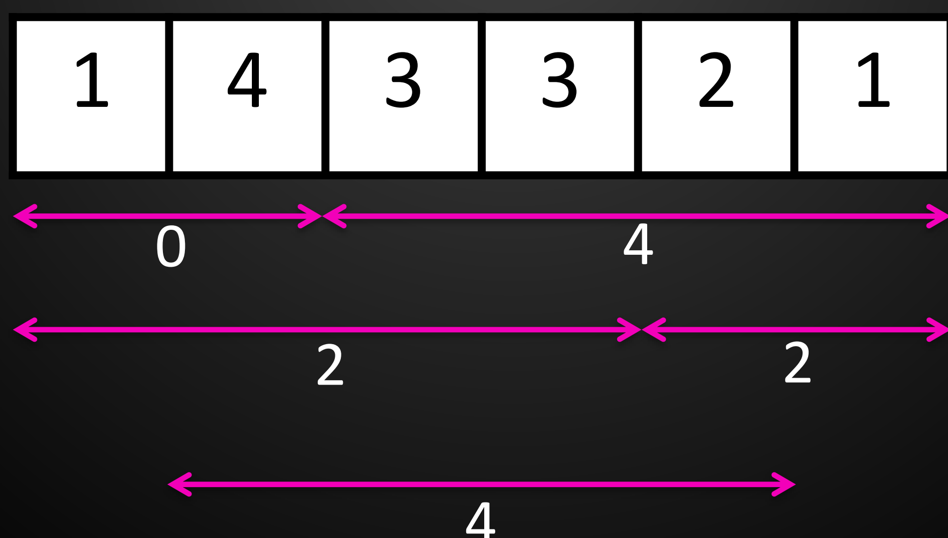
ダルマ落とし

- ★ 数列が A = a_0 a_1 … a_n 与えられる
- ★ 隣合った数字の差分が1以下のとき、それらの数字を消すことができる
  - ★ この操作を再帰的に繰り返すことができる
- ★ 最適の順序で数字を消したとき、消せる数字の個数の最大値を求めよ

- ★ $1 \leqq n \leqq 300$
- ★ $0 \leqq a_i \leqq 10^9$

DP for ダルマ落とし

- ★ dp[i][j]： i 番目から j 番目で落とせる最大値
  - ★ dp[i][j] = $\max_k$ (dp[i][k] + dp[k][j]) または、
  - ★ dp[i][j] = dp[i+1][j−1] + 2 (間の区間が全て消せる時)
  - ★ の最大値



コード for ダルマ落とし

短い区間から計算

区間の開始位置 i

```
// solve
int dp[n+1][n+1]={};
for(int k=1; k<=n-1; k+=2){
      for(int i=0; i+k<=n; i++){
            if( dp[i+1][i+k-1] == k-1 )
                  dp[i][i+k] = dp[i+1][i+k-1] + (abs(w[i]-w[i+k]) <= 1 ? 2 : 0);
            for(int j=i+1; j<=i+k-1; j+=2)
                  dp[i][i+k] = max(dp[i][i+k], dp[i][j] + dp[j+1][i+k] );
      }
}
// output
cout << dp[0][w.size()-1] << endl;
```

⚠全体の区間の長さが偶数になるように前処理してある
（奇数長のとき末尾にダミー要素を追加）