教科書輪講
プログラミングコンテスト攻略のためのアルゴリズムとデータ構造

第11章 動的計画法

東京工業大学 情報理工学院
情報工学系 知能情報コース
石田研究室 修士二年 曽我光瑛

目次

Gitがうまくいかない人のために

まずは状況を見る

今いるブランチの確認

```
$ git branch
* chapter11
  exercise
  master
```

今いるブランチでの作業状況の確認

```
$ git status
On branch chapter11
nothing to commit, working tree clean
```

基本は未保存の作業が残っていない状況で，
master ブランチに移動して pull (fetch + merge)する

状況１：masterブランチ以外でstatus が clean でない

- add + commit して作業状況を保存してmasterに移動する
- もし，今いるブランチにmaster にされた変更を取り入れたければ rebase または merge

Gitがうまくいかない人のために

状況2：masterに変更を加えてしまった（これは絶対に避けたほうがいい）

- commit する前
  - 今ある状況を別のブランチに移せばよい

```
$ git stash                          # ひとまず一時保存stackに退避
$ git checkout -b [branch_name]      # 新しいブランチに行く
$ git stash pop                      # 一時保存スタックから差分をpop
$ git add .                          # 新しいブランチで差分を履歴管理に
$ git commit -m "commit message"     # 新しい差分をコミットする
```

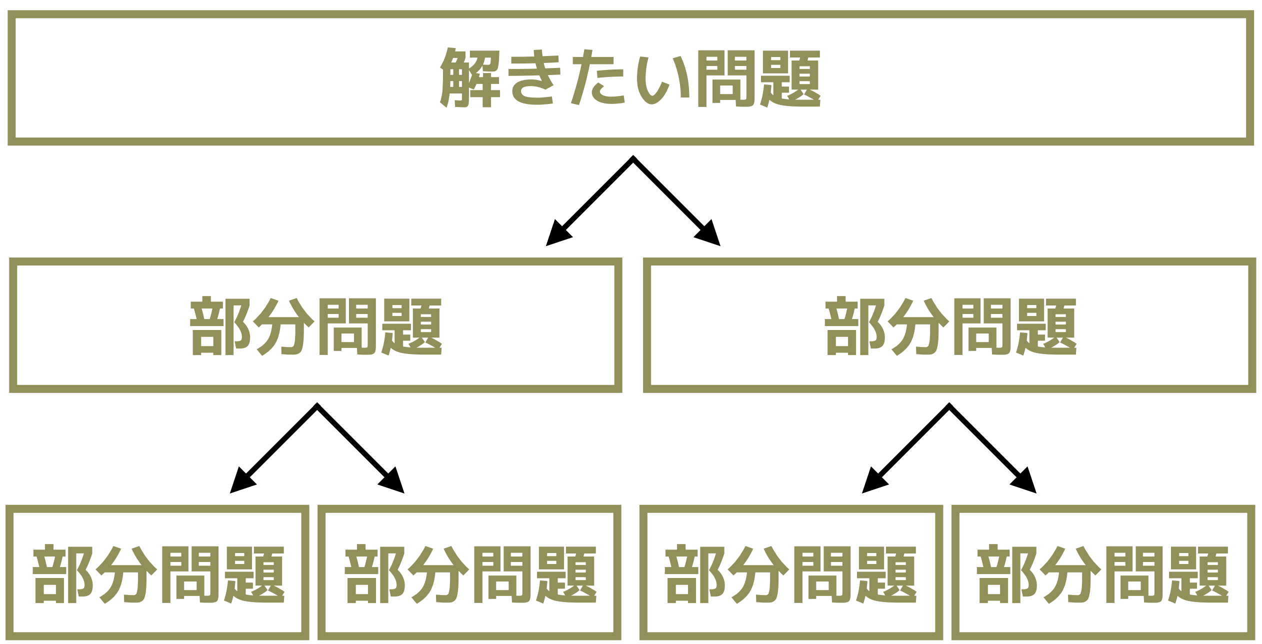
- commit した後
  - コミットする前に戻す（コミットした内容は消えてしまう）

```
$ git log                            # ログを表示してハッシュ値を確認
$ git reset --hard [commit_hash]     # コミットの取り消し
$ git pull                           # リモートからpull
```

  - reset --soft にすると作業ディレクトリの内容を残すことができるが，やったことないのでどうなるかわからん（もしかしたらstashに入れて別ブランチに移動できるかも）
- 時には諦めも重要（輪講程度のGitなんぞに時間を使うのはもったいない）

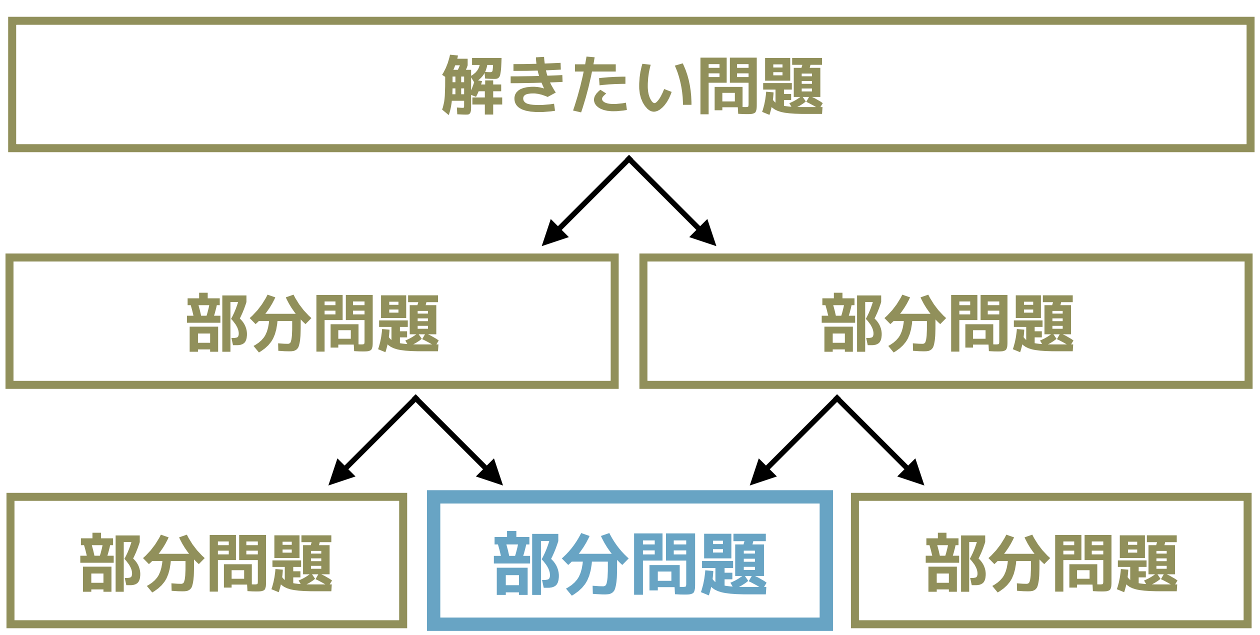
動的計画法(Dynamic Programming, DP)

問題を部分問題に分割して，最終的な解を得る手法



分割統治法

- 最終的に解を得たい問題を再帰的に部分問題に分割して，それらの解を統合することでもとの問題の解を得る
- 各部分問題は**独立**



動的計画法

- 再帰的に部分問題に分割するところは，分割統治法と考え方が一緒
- 各部分問題は**一部を共有**
- 共有する部分問題の値を保存することで，再計算の手間を省く

動的計画アルゴリズムの開発

動的計画法の代表的な適用例は**最適化問題**

1. 最適解の構造を特徴づける
2. 最適解の値を再帰的に定義する
3. （多くの場合はボトムアップ方式で）最適解の値を計算する
4. 計算された情報から1つの最適解を構成する

目次

11.1 Exhaustive Search

概要

長さnの数列**A**と整数mに対して，**A**の要素の中のいくつかの要素を足し合わせてmが作れるかどうかを判定するプログラムを作成してください．**A**の各要素は一度だけ使うことができます．
数列**A**が与えられた上で，質問としてq個のm_iが与えられるので，それぞれについて”yes”または”no"と出力してください．

入力

一行目にn，2行目に**A**を表すn個の整数，3行目にq，4行目にq個の整数m_iが与えられます．

出力

各質問について**A**の要素を足し合わせてm_iを作ることができればyesと，できなければnoと出力してください．

出典

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_A&lang=jp

Exhaustive Search の解とは…

要素数iの数列を，A_i = <$a_1, a_2, \ldots, a_i$>と表す.

Exhaustive Searchの解を求める関数を以下のように定式化する.

solve(A_n, m) = {和がmになる要素の組み合わせがA_nに存在する}

ここで，solve()を再帰的に定義すると,

solve(A_n, m) = solve(A_{n-1}, m) or solve(A_{n-1}, m - a_n)

Exhaustive Search の実装

コードに落とし込む

```
#include <iostream>
#include <string.h>
#define MAX_N 20
#define MAX_SUM 40000
using namespace std;
bool dp[MAX_N+1][MAX_SUM+1];
bool checked[MAX_N+1][MAX_SUM+1];
int a[MAX_N];
bool solve(int n, int m) {
	if (checked[n][m]) return dp[n][m];

	if (m == 0) {
		return true;
	} else if (n == 0) {
		return false;
	}
	checked[n][m] = true;
	return dp[n][m] = solve(n-1, m) || solve(n-1, m - a[n-1]);
}

int main() {
	int n, q, m;
	cin >> n;
	for (int i = 0; i < n; i++) {
		cin >> a[i];
	}
	cin >> q;
	memset(dp, 0, sizeof(dp));
	memset(checked, 0, sizeof(checked));
	int input;
	for (int i = 0; i < q; i++) {
		cin >> input;
		if (solve(n, input)) {
			cout << "yes" << endl;
		} else {
			cout << "no" << endl;
		}
	}
	return 0;
}
```

計算されていたらその値を再利用

再帰の基底状態の定義

値を保存して再帰

俗に言う**メモ化再帰**というやつ

実際は，dp[n][m]の値について

-1 -> 未計算

0 -> false

1 -> ture

のように定義すればchecked[][]は不要．

今回はわかりやすさのために別の配列を定義した

ちょっと真面目に考える

実はメモ化再帰によるDPじゃなくてもできる

メモ化再帰でやったこと

- 下の表を**トップダウン的**に埋めていく
- 埋めた値は再利用

aの何個目の数字まで使うか

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T																	
1																		
2	T							?										
3								?										?
4																		?
5																		?

以下の数列a のいくつかの要素を選んで，和がmになるか調べる

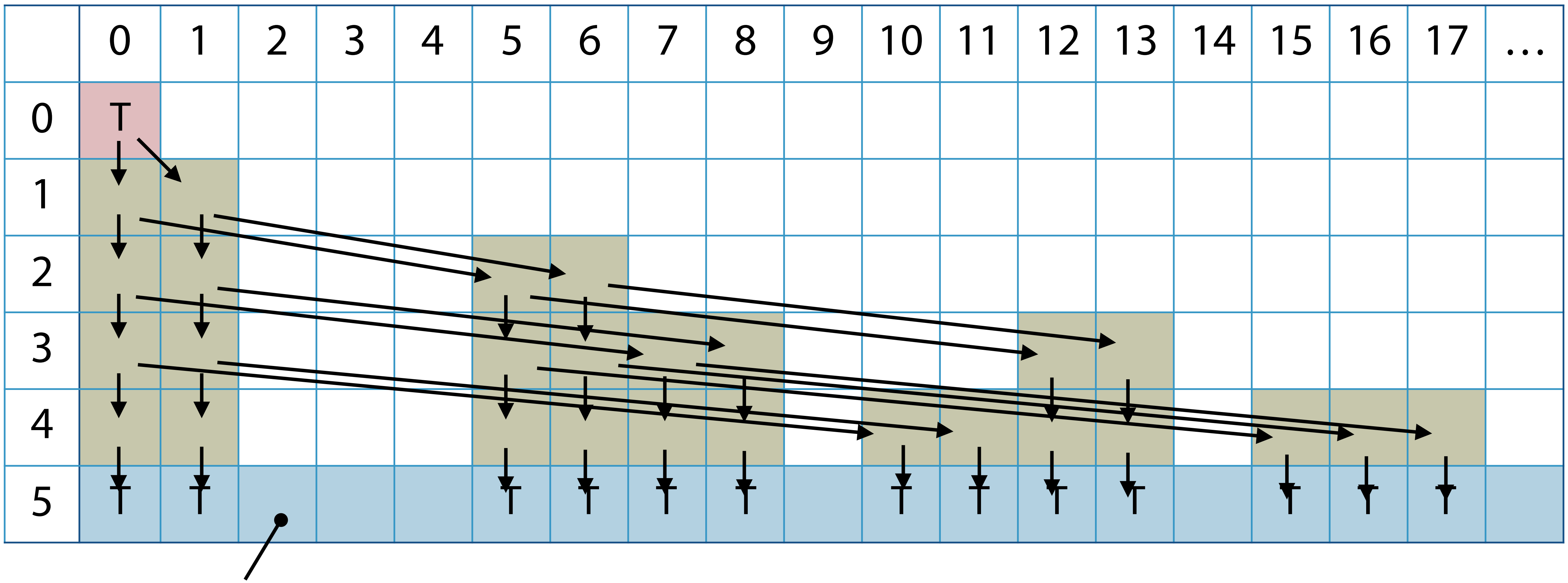
a = {1, 5, 7, 10, 21}

m = 17

ボトムアップ的にDPテーブルを埋める

以下の数列a のいくつかの要素を選んだ和の部分を塗りつぶしていく(一部略)

a = {1, 5, 7, 10, 21}



dp[5][m]が表すもの

5個の要素からいくつか選んで和がmになるか？

実装方針

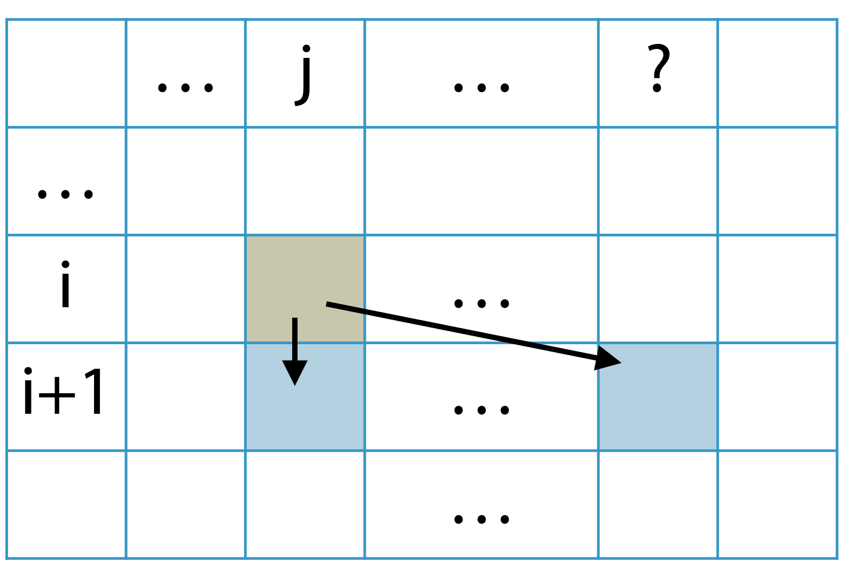
DPテーブルを全部埋めて，dp[5][m]がtrueか調べれば良い（計算量O(mn)）

実装してみよう（演習：ex0.cpp）


```
/* loop.cpp */

void solve(int n) {
   dp[0][0] = 1;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < MAX_SUM; j++) {

         ここで行われているdp[][]の更新処理
         を実装してください

      }
   }
}
```

補足
数列はa[]とする

ヒント



i 行 j 列のtrue/falseは，
i+1 行目のどの要素のtrue/false に
どのように影響を与える？

実装例


```cpp
/* loop.cpp */

void solve(int n) {
   dp[0][0] = 1;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < MAX_SUM; j++) {
         dp[i+1][j] |= dp[i][j];
         dp[i+1][j+a[i]] |= dp[i][j];
         }
      }
   }
}
```

考察と余談

計算量

n ： 入力される点数の種類（リストの長さ）

m ： 入力の総和の最大値

	時間計算量	空間計算量
再帰	$O(2^n)$	O(n)
メモ化再帰	O(mn)	O(mn)
ループ	O(mn)	O(mn)

余談

これと似たような問題（例．0-1ナップザック問題）などは，いつでもDPで良いかというと違う

例えば，n = 20， m = 10^9の場合を考えてみると，

$2^{20} = 1{,}048{,}576 ≒ 10^6$

$mn = 20{,}000{,}000{,}000 = 2 * 10^{10}$

いつでもDP思想は危険な場合がある．

（メモ化再帰の場合は関数呼び出しが高々2^nなので，実はそんなに変わらない）

目次

11.2 フィボナッチ数列

概要

フィボナッチ数列の第n項の値を出力するプログラムを作成してください．ここではフィボナッチ数列は以下の再帰的な式で定義します．

$$fib(n) = \begin{cases} 1 & (n = 0) \\ 1 & (n = 1) \\ fib(n-1) + fib(n-2) & \end{cases}$$

入力

1つの整数nが与えられます

出力

フィボナッチ数列の第n項の値を出力してください

出典

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_10_A

実装

再帰による実装

```c
int fibonacci(int n) {
    if (n < 2) return 1;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

メモ化再帰

```c
#define MAX_N 44
int dp[MAX_N+1];
int fibonacci(int n) {
    if (dp[n] > 0) return dp[n];

    if (n < 2) {
        return dp[n] = 1;
    } else {
        return dp[n] = fibonacci(n-1) + fibonacci(n-2);
    }
}
```

ループ（DP）

```c
#define MAX_N 44
int dp[MAX_N+1];
int fibonacci(int n) {
    if (n < 2) {
        return 1;
    } else {
        dp[0] = dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i-1] + dp[i-2];
        }
    }
    return dp[n];
}
```

目次

11.3 最長共通部分列

概要

最長共通部分列問題（**L**ongest **C**ommon **S**ubsequence : **LCS**）は，2つの与えられた列$X = \{x_1, x_2, \ldots, x_m\}$ と $Y = \{y_1, y_2, \ldots, y_n\}$ の最長共通部分列を求める問題です.
与えられた2つの文字列X, Yに対して，最長共通部分列Zの長さを出力するプログラムを作成してください.
与えられる文字列は英文字のみで構成されています.

入力

複数のデータセットが与えられます．最初の行にデータセットの数qが与えられ，続く2q 行にデータセットとして文字列X, Yがそれぞれ一行に与えられます

出力

各データセットについてX, Yの最長共通部分列Zの長さを1行に出力してください

出典

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_10_C

解き方を考えよう

みんな大好き配列アラインメントを求める問題
まずは最適解の特徴付けから

$X = \langle x_1, x_2, \ldots, x_m \rangle$ と $Y = \langle y_1, y_2, \ldots, y_n \rangle$ を列，$Z = \langle z_1, z_2, \ldots, z_k \rangle$ を XとYの任意のLCSとする

1. $x_m = y_n$ ならば $z_k = x_m = y_n$ でありZ$_{k-1}$ は X_{m-1} と Y_{n-1} のLCSである
2. $x_m \neq y_n$ のとき，$z_k \neq x_m$ ならば Z は X_{m-1} と Y のLCSである
3. $x_m \neq y_n$ のとき，$z_k \neq y_n$ ならばZはXとY_{n-1}のLCSである

証明

1. $x_m = y_n$ で，$x_m \neq z_k$ である場合を考えて矛盾を導く
2. 上の仮定で Z は X_{m-1} と Y の共通部分列であるが， Y と X_{m-1} との間に長さk+1 以上の共通部分列が存在すると仮定するとZが長さkのLCSであることに矛盾する
3. 2と対称に考える

最適解を再帰的に定義

解くべき部分問題

- $x_m = y_n$ のときに
  - X_{m-1} と Y_{n-1} からLCSを見つける

```
itisatruthuniversallyacknowle
                             |
alicewasbeginningtogetverytire
```

- xm ≠ yn のとき
  - X_{m-1} と Y_n のLCSを見つける

```
itisatruthuniversallyacknowl
                            |
alicewasbeginningtogetverytir
```

  - X_m と Y_{n-1} のLCSを見つける

```
itisatruthuniversallyacknowl
                            |
alicewasbeginningtogetverytir
```

共通する部分列（メモ化する意味がある）

定式化

c[i][j]を，XiとYjのLCSの長さとする

$$c[i][j] = \begin{cases} 0 & \text{if i = 0 or j = 0} \\ c[i-1][j-1]+1 & \text{if i, j > 0 and } x_i = y_i \\ max(c[i][j-1], c[i-1][j]) & \text{if i, j > 0 and } x_i \neq y_i \end{cases}$$

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			m	**i**	**s**	**h**	**i**	**m**	**a**	**g**	**a**	**k**	**u**	**c**	**h**	**o**	**u**
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	**p**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	**i**	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	**k**	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2
4	**a**	0	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2
5	**c**	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3
6	**h**	0	0	1	1	2	2	2	2	2	2	2	2	3	4	4	4
7	**u**	0	0	1	1	2	2	2	2	2	2	2	3	3	4	4	5

実装


```
#include <iostream>
#include <string>
#include <string.h>
using namespace std;
#define SIZE 1000

int c[SIZE+1][SIZE+1];

void lcs(string str_x, string str_y) {
    for (int i = 1; i <= str_x.length(); i++) {
        for (int j = 1; j <= str_y.length(); j++) {
            if (str_x[i-1] == str_y[j-1]) {
                c[i][j] = c[i-1][j-1] + 1;
            } else {
                c[i][j] = max(c[i][j-1], c[i-1][j]);
            }
        }
    }
}

int main() {
    int n;
    cin >> n;
    string str_x, str_y;
    while (cin >> str_x >> str_y) {
        lcs(str_x, str_y);
        cout << c[str_x.length()][str_y.length()] << endl;
        memset(c, 0, sizeof(c));
    }
    return 0;
}
```

計算量

O(nm)

LCSを出力する

		m	i	s	h	i	m	a	g	a	k	u	c	h	o	u
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
k	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2
a	0	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2
c	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3
h	0	0	1	1	2	2	2	2	2	2	2	2	3	4	4	4
u	0	0	1	1	2	2	2	2	2	2	2	3	3	4	4	5

移動規則

1. 文字が一致している ↖
2. 今見ている文字と同じ数値の方向へ進む ← ↑

発見されたLCS　i a c h u
i k c h u

目次

概要

n 個の行列の連鎖M_1, M_2, M_3, …, M_n が与えられた時，スカラー乗算の回数が最小になるように積 $M_1M_2M_3 \cdots M_n$ の計算順序を決定する問題を連鎖行列積問題といいます.

n 個の行列について，行列 M_i の次元が与えられた時，積$M_1M_2 \cdots M_n$の計算に必要なスカラー乗算の最小の回数を求めるプログラムを作成してください.

入力

入力の最初の行に，行列の数 n が与えられます．続く n 行で行列M_i($i = 1 \cdots n$)の次元が空白区切りの2つの整数 r, c で与えられます．r は行列の行の数，c は行列の列の数を表します.

出力

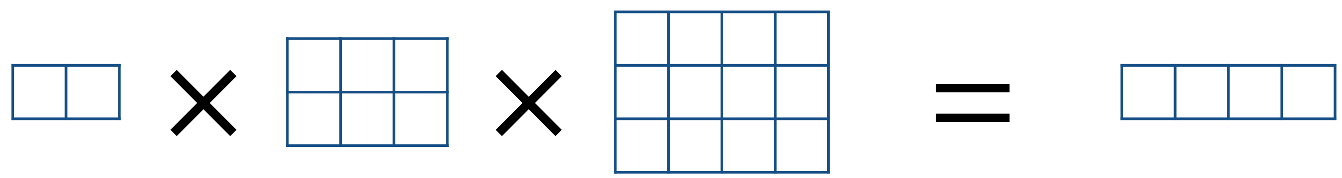
最小の階数を一行に出力してください.

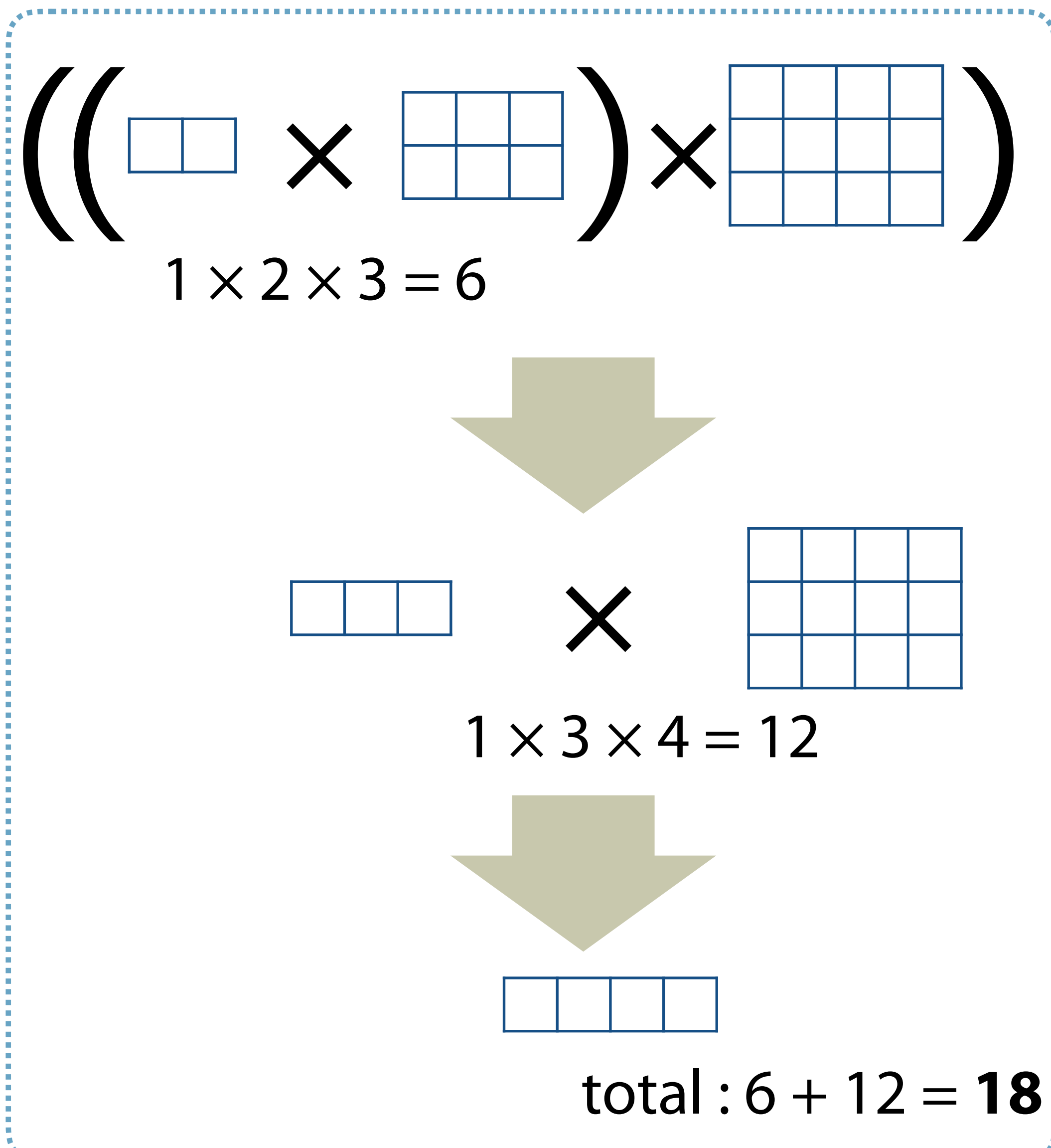
出典

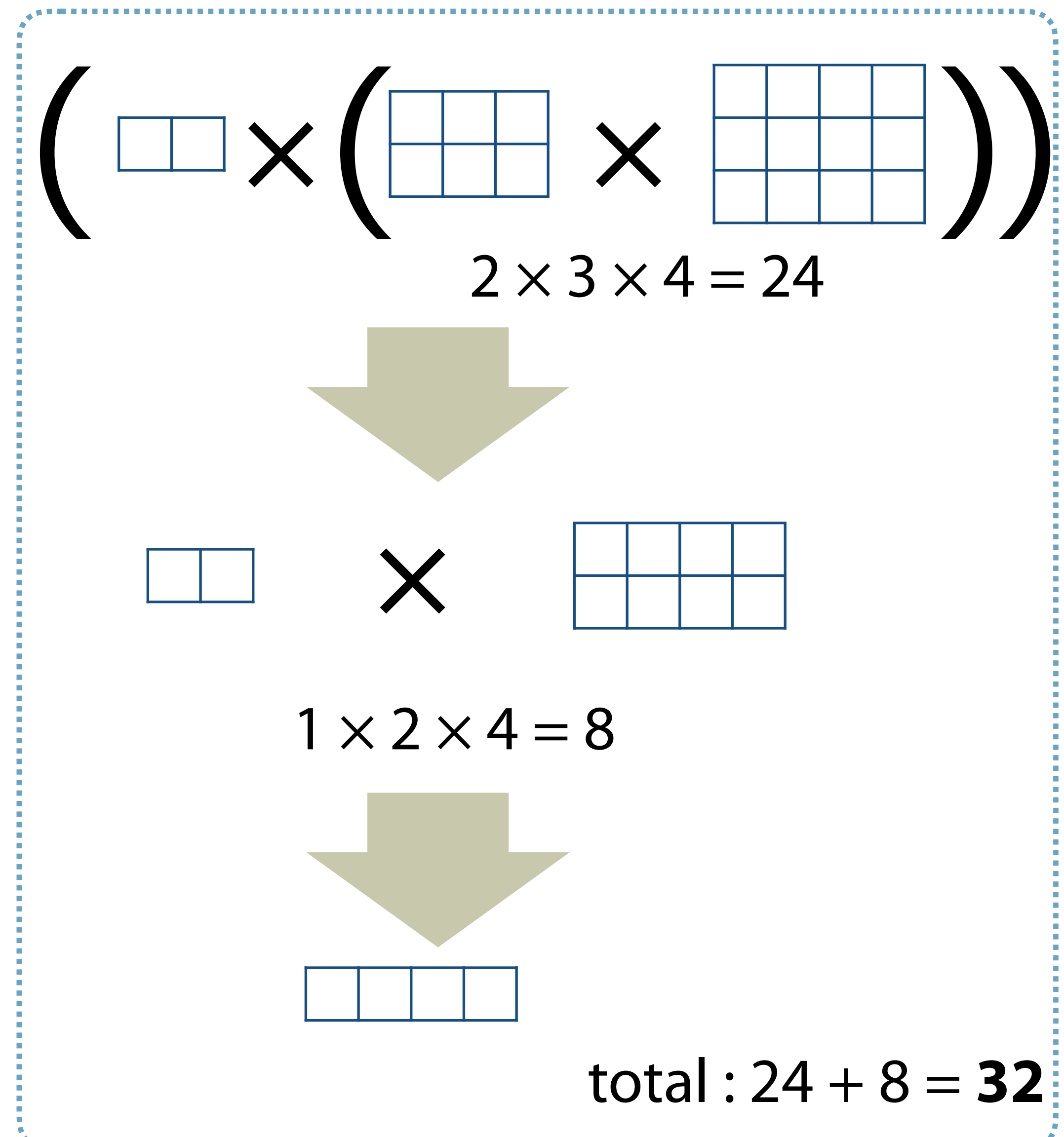
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_10_B

連鎖行列積問題

行列積を求めるときのスカラー乗算の回数の最小化







最適解の特徴を考える

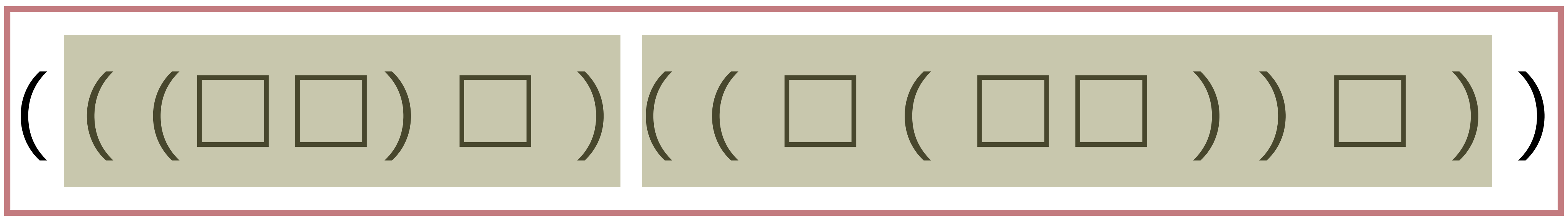
n 個の行列 <A_1, A_2, …, A_n> の行列積を$A_{1 \ldots n}$ と表す．

いま，$A_{i \ldots j}$について，$i \leqq k < j$ を満たすk で積として分割する．すなわち，

$A_{i \ldots j}$ を $A_{i \ldots k}$ と $A_{k+1 \ldots j}$ の積として分割する．

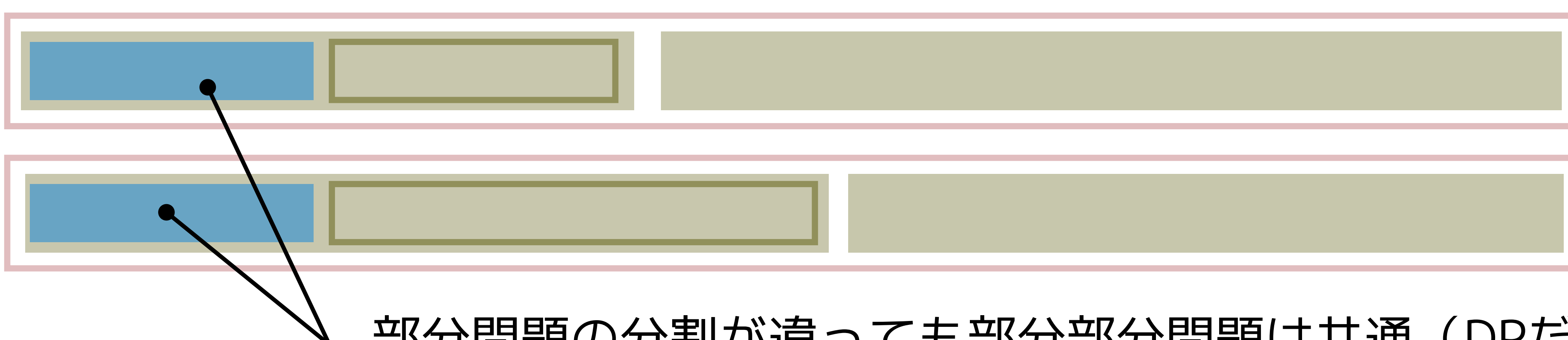
このとき，$A_{i \ldots j}$ の最適括弧付けが $A_{i \ldots k}$ と $A_{k+1 \ldots j}$ に分割できる時

$A_{i \ldots k}$ は <A_i, A_{i+1}, …, A_k>に対する最適な括弧付けでなければならない．



部分の括弧付けが最適なら，

部分の括弧づけは最適である（部分構造最適性（DPかな？））



部分問題の分割が違っても部分部分問題は共通（DPだ）

解を再帰的に定義

$A_{1...n}$ の最適括弧付けをした時のスカラ乗算の計算回数をm[i][j]とおく．
連鎖行列の i 番目の行列の行数を p_{i-1}，列数を p_i と表すとき m[i][j] は
以下のように定義される．ただし i ≦ k < j とする．

$$m[i][j] = \begin{cases} 0 & \text{i = j の時} \\ \min_{i \leq k \leq j}(m[i][j], m[i][k] + m[k+1][j] + p_{i-1} * p_k * p_j) & \text{i < j の時} \end{cases}$$

m[i][j] は区間[i, j]を含むような場合を計算するときに再利用される

実装例（メモ化再帰）


```
#include <iostream>
#include <climits>
#include <string.h>

using namespace std;

static const int N = 100;
int mcm[N+1][N+1], p[N+1];

int calc_mcm(int i, int j) {
	if(mcm[i][j] >= 0) {
		return mcm[i][j];
	}
	if(i == j) {
		return mcm[i][j] = 0;
	}else {
		int ans = INT_MAX;
		for (int k = i; k < j; k++) {
			ans = min(ans, calc_mcm(i, k) + calc_mcm(k+1, j) + p[i-1] * p[k] * p[j]);
		}
		return mcm[i][j] = ans;
	}
}

void solve() {
	int n, row, col, i;
	cin >> n;
	i = 0;
	while (++i <= n) {
		cin >> row >> col;
		p[i-1] = row;
		p[i] = col;
	}
	memset(mcm, -1, sizeof(mcm));
	cout << calc_mcm(1, n) << endl;
}


int main() {
	solve();
	return 0;
}
```

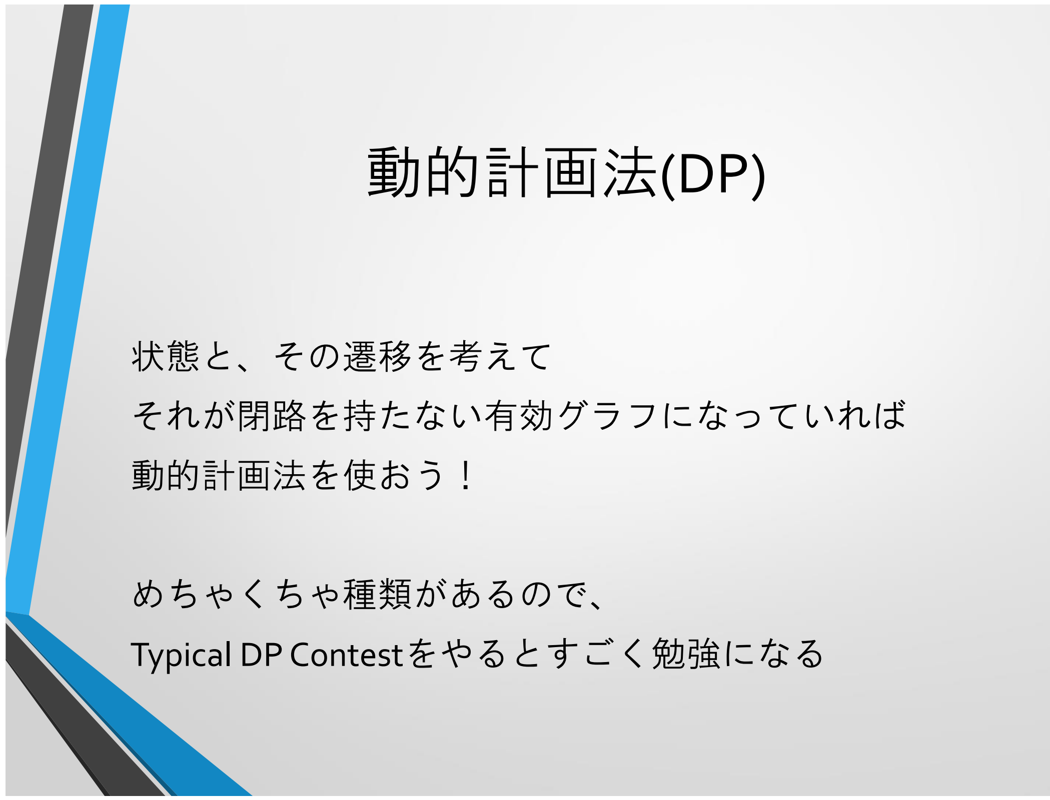
実装例（ループ）


```cpp
#include <iostream>
#include <climits>
#include <string.h>

using namespace std;

static const int N = 100;
int mcm[N+1][N+1], p[N+1];

void calc_mcm(int n) {
	for (int i = 1; i <= n; i++) {
		mcm[i][i] = 0;
	}
	for (int num_matrix = 2; num_matrix <= n; num_matrix++) {
		for (int i = 1; i <= n - num_matrix + 1; i++) {
			int j = i + num_matrix - 1;
			mcm[i][j] = INT_MAX;
			for (int k = i; k < j; k++) {
				mcm[i][j] = min(mcm[i][j], mcm[i][k] + mcm[k+1][j] + p[i-1] * p[k] * p[j]);
			}
		}
	}
}

void solve() {
	int n, row, col, i;
	cin >> n;
	i = 0;
	while (++i <= n) {
		cin >> row >> col;
		p[i-1] = row;
		p[i] = col;
	}
	calc_mcm(n);
	cout << mcm[1][n] << endl;
}


int main() {
	solve();
	return 0;
}
```

目次

蛇足 | DPを使う場面とは

参考資料の引用だけですが…



秋山研究室OB　小峰駿汰氏のLTスライドより引用

http://d.hatena.ne.jp/Tayama/20111210/1323502092

・ DPはDAGであると説明しているサイト

http://d.hatena.ne.jp/komiyam/20111212/1323615687

・ DPのメリットとかを解説してくれています

目次

動的計画法とは

本編（教科書解説）
  11.1 Exhaustive Search
  11.2 フィボナッチ数列
  11.3 最長共通部分列
  11.4 連鎖行列積

蛇足

演習問題

Typical DP Contest

概要

- DPの練習をすることが目的で作られたコンテスト
- 競プロ界隈（主に小峰さん）が推してきた
- 問題の順番は必ずしも難易度順ではない（らしい）
- この中からいくつか解いてみる
- 楽しくない（楽しい）

出典

http://tdpc.contest.atcoder.jp/

A. コンテスト

Problem Statement

N 問の問題があるコンテストがあり、i 問目の問題の配点は p_i 点である。コンテスタントは、この問題の中から何問か解き、解いた問題の配点の合計が得点となる。このコンテストの得点は何通り考えられるか。

Constraints

- $1 \leq N \leq 100$
- $1 \leq p_i \leq 100$

Input Format

入力は以下の形式で標準入力から与えられる。

N
$p_1 p_2 \dots p_N$

Output Format

答えを一行に出力せよ。

解き方を考えてみる

Exhaustive Search を思い出してみる

Exhaustive Search

数列A内のいくつかの要素の和がある整数mとなるものは存在するか

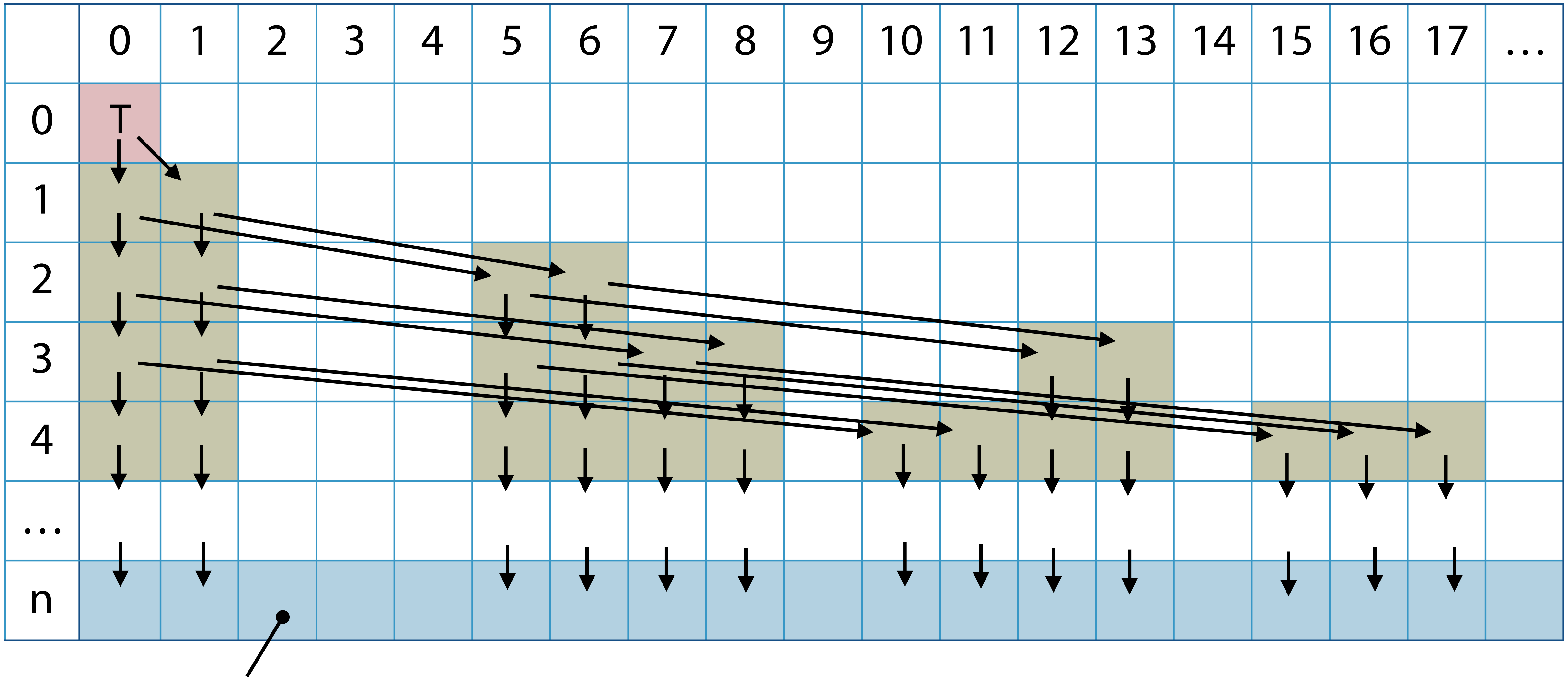
今回の問題

数列A内のいくつかの要素の和は何通りあるか

頭空っぽにして．．．

Exhaustive Search のDPテーブル全部埋めて，dp[n][*]でtrueになってるところ数え上げればよくね？

もうちょっと真面目に考える（ボトムアップでの解法）



dp[n][m]が表すもの
n個の要素からいくつか選んで和がmになるか？

実装方針

DPテーブルを全部埋めて，dp[n][*]のTrueの個数を数えればよい
（計算量：テーブル構成 -> O(mn)，数え上げ -> 定数）

演習問題：ex1.cpp

is_correct(n, m)の実装

- 問題をn問目まで解いて，得点がmになるか
- exhaustive_searchの亜種みたいなやつ
- solve_recursive()の中で使われている
- dp[n][m]
  - -1 (未探索)
  - 0 (得点がmになる組み合わせがない)
  - 1 (得点がmになる組み合わせが存在する)
- p[n]
  - n-1問目の問題の得点

参考（になるか不明だけど）までにループでdpテーブル埋める実装も一緒に載っけてる

```cpp
#include <iostream>
#include <string.h>
using namespace std;
static const int N = 101;
static const int MAX_SUM = 10001;

int dp[N][MAX_SUM];
int p[N];
int n;

void solve() {
    dp[0][0] = 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < MAX_SUM; j++) {
            dp[i+1][j] |= dp[i][j];
            dp[i+1][j+p[i]] |= dp[i][j];
        }
    }
    int ans = 0;
    for (int i = 0; i < MAX_SUM; i++) {
        ans += dp[n][i];
    }
    cout << ans << endl;
}

int is_correct(int n, int m) {
    /**
     * ここに実装
     */
}

void solve_recursive() {
    memset(dp, -1, sizeof(dp));
    int ans = 0;
    for (int i = 0; i < MAX_SUM; i++) {
        if (is_correct(n, i) > 0) {
            ans++;
        }
    }
    cout << ans << endl;
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> p[i];
    }
    // solve()
    solve_recursive();
    return 0;
}
```

追加課題（暇な人用）

2017年6月3日に行われた AtCoder Regular Contest 075 の一問目
ここまでの内容をやっておけば，問題文を見た瞬間に解法が1つ浮かぶはず

問題文

あなたはコンピュータで試験を受けています。試験は N 問の問題からなり、i 問目の問題の配点は s_i です。それぞれの問題に対するあなたの解答は「正解」または「不正解」のいずれかとして判定され、正解した問題の配点の合計があなたの成績となります。あなたが解答を終えると、解答がその場で採点されて成績が表示される…はずでした。

ところが、試験システムに欠陥があり、成績が 10 の倍数の場合は、画面上で成績が 0 と表示されてしまいます。それ以外の場合は、画面に正しい成績が表示されます。この状況で、成績として画面に表示されうる最大の値はいくつでしょうか？

制約

- 入力値はすべて整数である。
- $1 \leq N \leq 100$
- $1 \leq s_i \leq 100$

入力

入力は以下の形式で標準入力から与えられる。

$$
\begin{array}{l}
N \\
s_1 \\
s_2 \\
: \\
s_N
\end{array}
$$

出力

成績として画面に表示されうる最大の値を出力せよ。

出典：http://arc075.contest.atcoder.jp/tasks/arc075_a

B. ゲーム

Problem Statement

すぬけ君とすめけ君がゲームをしている。最初に、二つの山がある。左の山には A 個の物が積まれており、上から i 番目のものの価値は a_i である。左の山には B 個の物が積まれており、上から i 番目のものの価値は b_i である。すぬけ君とすめけ君は、すぬけ君からはじめて交互に次の操作を繰り返す。

- 両方の山が空であれば、ゲームを終了する。
- 片方の山が空であれば、空でないほうの山の一番上のものをとる。
- 両方の山が空でなければ、好きなほうの山を選び、一番上のものをとる。

両者が最善を尽くしたとき、すぬけ君の取るものの価値の合計を求めよ。

Constraints

- $1 \leq A, B \leq 1000$
- $1 \leq a_i, b_i \leq 1000$

Input Format

入力は以下の形式で標準入力から与えられる。

$A \quad B$
$a_1 \ldots a_A$
$b_1 \ldots b_B$

Output Format

答えを一行に出力せよ。

”最適な行動”をどう表現するか？

ゲームの状況はs1 と山札の残り枚数によって完全に再現できる

- p2 の得点 s2 はカードの総得点から s1 を引いたものである
- 一度引いたカードは再利用されない
- 今回計算したいのはs1 なので，実質ゲームの状況とs1は等価

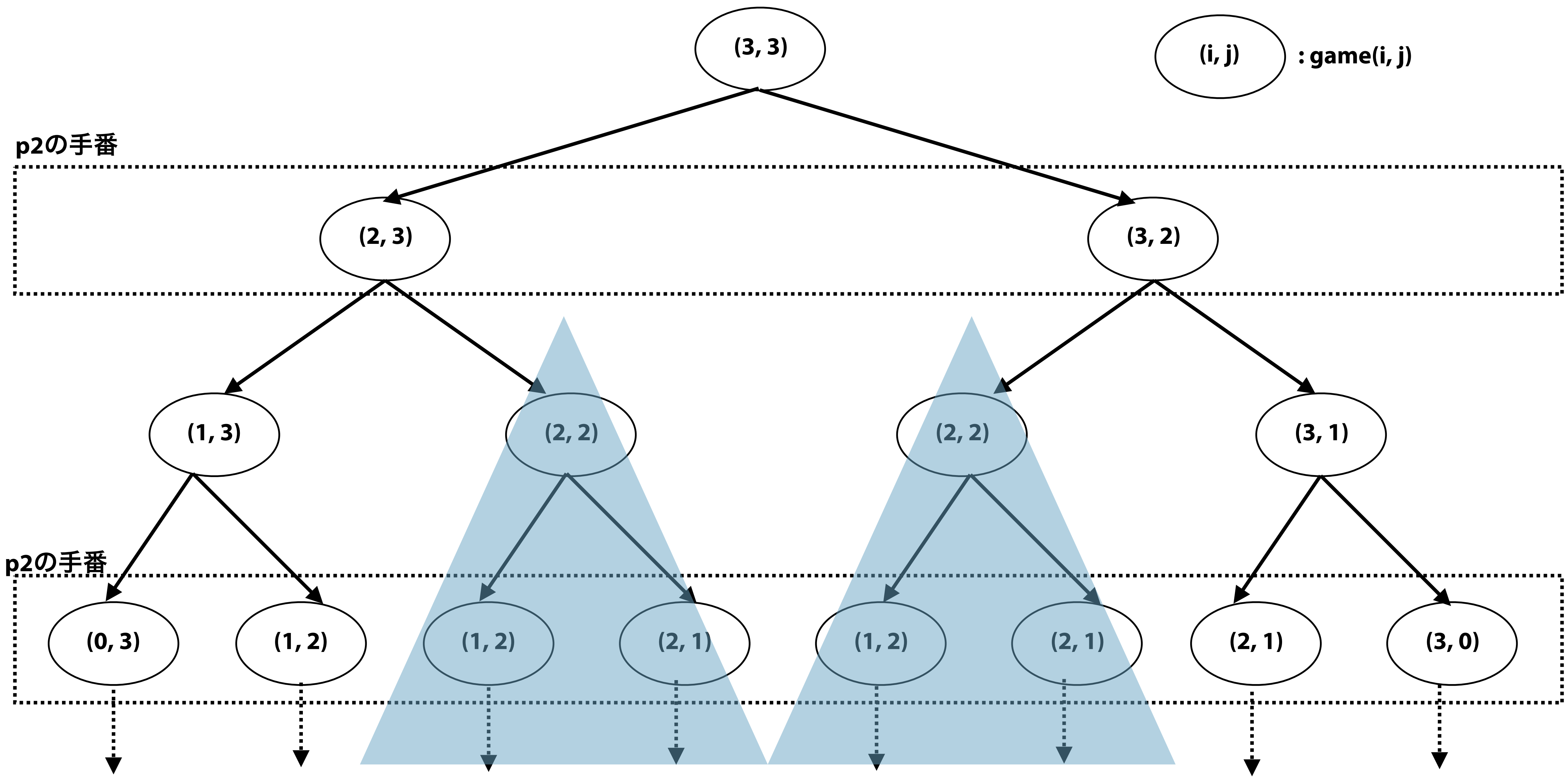
二人のプレイヤーp1, p2 のp1の得点s1に対する行動について以下が成り立つ

- p1 は s1 がより大きくなるように山札を選ぶ
- p2 は s1 がより小さくなるように山札を選ぶ

s1 を表すgame(i, j) を以下のように定義する．ただし，a, bはそれぞれ山札a, bとする

- game(i, j) = 0 (i = j = 0 の時)
- game(i, j) = max(game(i-1, j) + a[i], game(i, j-1) + b[j]) (p1の手番の時）
- game(i, j) = min(game(i-1), j), game(i, j-1))（p2の手番の時）

どうやって解く？



同じ部分問題が出てきたのでDPで行ける!!

- ゲームの問題は状態遷移を考えて閉路のないグラフになれば DPを考えてみる価値あり（ターン制のゲームはほぼDPしか出会ったことがない）
- 入力的に線形じゃないと落ちそうなゲームは頑張ってヒューリスティックを探す

演習問題：ex2.cpp

ほぼ穴埋めで解けるはず…

dp[i][j]

- 山Aの残り枚数がi 枚，山Bの残り枚数がj 枚のときのすぬけ君の得点の最大値

deck_a[i]/deck_b[j]

- それぞれA/Bの山の上からi/j番目のカードの得点

a/b

- A/B の山のカード枚数

```cpp
#include <iostream>
#include <string.h>
using namespace std;

static const int CARD_MAX = 1001;

int dp[CARD_MAX][CARD_MAX];
int deck_a[CARD_MAX], deck_b[CARD_MAX];
int a, b;

int game(int i, int j) {
    if (i + j == 0) {
        return 0;
    }
    if (~dp[i][j]) {
        return dp[i][j];
    }

    if (1 - ((a + b) - (i + j)) % 2) {
        /**
         * ここにすぬけ君の手番のときの処理を書く
         */
    } else {
        /**
         * ここにすめけ君の手番のときの処理を書く
         */
    }
}

int main() {
    cin >> a >> b;
    for (int i = 0; i < a; i++) {
        cin >> deck_a[a - i];
    }
    for (int i = 0; i < b; i++) {
        cin >> deck_b[b - i];
    }
    memset(dp, -1, sizeof(dp));
    cout << game(a, b) << endl;
    return 0;
}
```

Problem Statement

2^K 人が参加するトーナメントがある。このトーナメントでは以下の形式で試合を行う。

- 第 1 ラウンドでは、1 と 2、3 と 4、… が試合を行う。
- 第 2 ラウンドでは、(1 と 2 の勝者) と (3 と 4 の勝者), (5 と 6 の勝者) と (7 と 8 の勝者), … が試合を行う。
- 第 3 ラウンドでは、((1 と 2 の勝者) と (3 と 4 の勝者) の勝者) と ((5 と 6 の勝者) と (7 と 8 の勝者) の勝者), ((9 と 10 の勝者) と (11 と 12 の勝者) の勝者) と ((13 と 14 の勝者) と (15 と 16 の勝者) の勝者), … が試合を行う。
- 以下同様に第 K ラウンドまで行う。

第 K ラウンドの終了後に優勝者が決定する。人 i の Elo Rating が R_i であるとき、人 i の優勝確率を求めよ。

ただし、Elo Rating R_P の人 P と Elo Rating R_Q の人 Q が対戦した場合、人 P が勝つ確率は $1/(1+10^{(R_Q-R_P)/400})$ であり、異なる試合の勝敗は独立であるとする。

Constraints

- $1 \le K \le 10$
- $0 \le R_i \le 4000$

Input Format

入力は以下の形式で標準入力から与えられる。

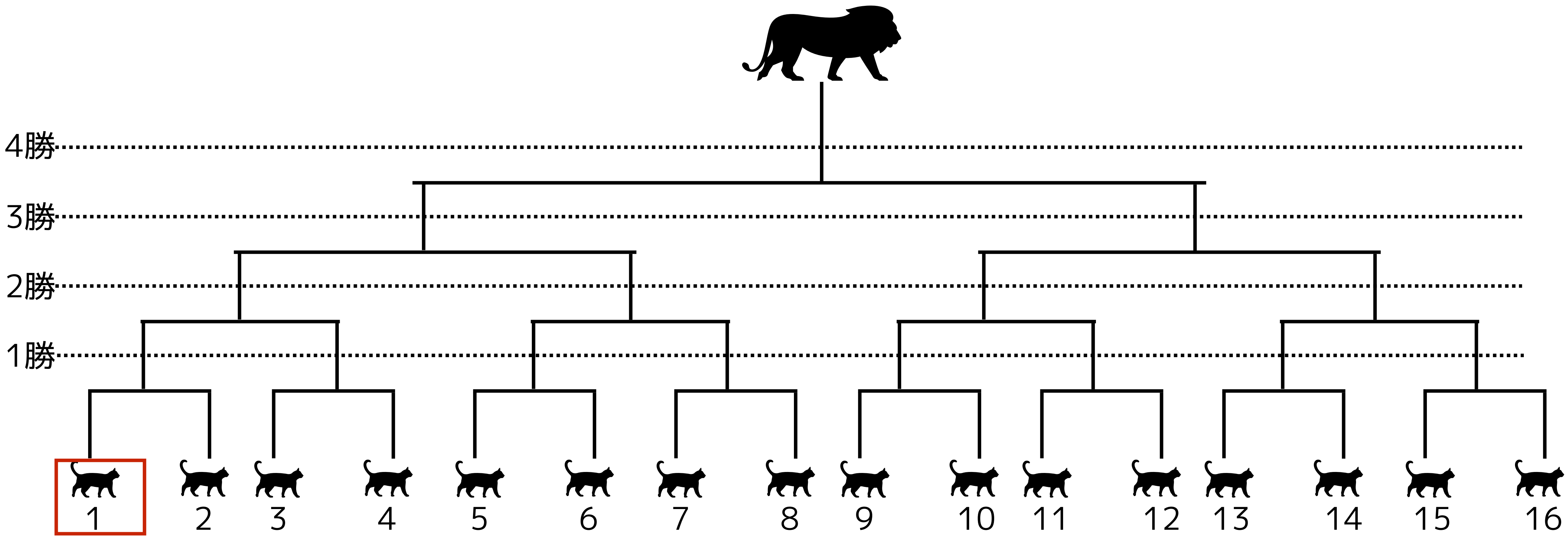
K
R_1
$\dots$
R_{2^K}

Output Format

答えを 2^K 行出力せよ。i 行目は人 i が優勝する確率であり、絶対誤差が 10^{-6} 以下のとき正当と判定される。

解法を考える

あまり難しく考えない



1がこのトーナメントで優勝するためには4回勝たなければならない

1. 隣にいる2を倒す
2. 隣のブロックから勝ち上がってきた3 or 4 を倒す
3. 隣のブロックから勝ち上がってきた5 or 6 or 7 or 8 を倒す
4. 隣のブロックから勝ち上がってきた9 or 10 or ... or 16 を倒す

では確率はどう考える？

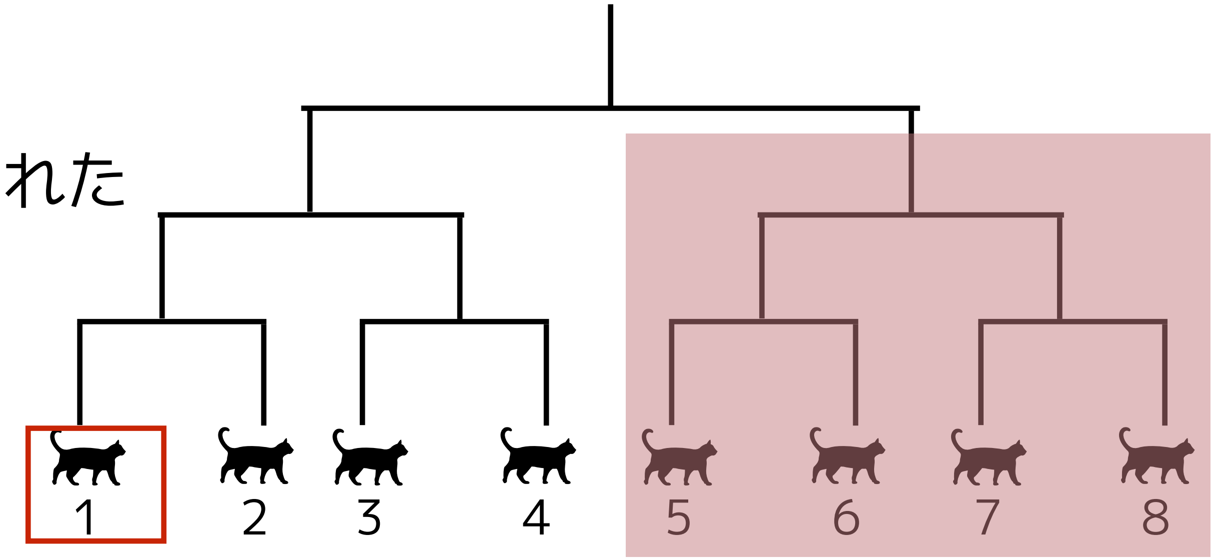
プレイヤーが勝つ確率

p(i, k)：プレイヤー i が k 回勝つ確率
w(i, j)：プレイヤー i が プレイヤー j に勝つ確率
E：k 回目に対戦することになるブロックのプレイヤー集合

$$p(i,k) = p(i,k-1)\sum_{e \in E} p(e,k-1)w(i,e)$$

k回目で対戦する相手をどう判定する？

- 自分がどちらのブロックに属しているかで条件分岐
- 自分が属していない方のブロックの
  プレイヤーと対戦すればよい
- 右の図で，1が 3 回目に闘うのは色付けされた
  領域にいるプレイヤーたち



演習問題：ex3.cpp

solve()で呼び出すwq(i, k)を完成させてください.
ただし，メモ化再帰で実装してください

ratings[i]
- プレイヤー i のレート

p[i][k]
- プレイヤー i が トーナメントで k 回勝つ確率

win(p1, p2)
- プレイヤー p1 が p2 に勝つ確率

```cpp
#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;
static const int PLAYERS = 1025;
static const int ROUND = 11;

double ratings[PLAYERS];
double p[PLAYERS][ROUND];
int n;

double win(int p1, int p2) {
    return 1.0 / (1.0 + pow(10, (ratings[p2] - ratings[p1])/400.0));
}

double wp(int i, int k) {
    if (p[i][k] > 0) return p[i][k];
    if (k == 0) return p[i][k] = 1.0;
    /**
     * ここに，メモ化再帰でこの問題を解くときのコードを書く
     */
    return p[i][k];
}

void solve_by_loop() {
    for (int i = 0; i < (int)pow(2.0, n); i++) {
        p[i][0] = 1.0;
    }
    for (int k = 1; k <= n; k++) {
        for (int i = 0; i < (int)pow(2.0, n); i++) {
            int index = i / (int)pow(2.0, k-1);
            int player;
            if (index % 2 == 0) {
                player = (index+1)*pow(2,k-1);
            } else {
                player = (index-1)*pow(2,k-1);
            }
            for (int j = 0; j < pow(2, k-1); j++) {
                p[i][k] += p[i][k-1]*win(i, player+j)*p[player+j][k-1];
            }
        }
    }
    for (int i = 0; i < pow(2, n); i++) {
        printf("%.10lf\n", p[i][n]);
    }
}
void solve() {
    for (int i = 0; i < (int)pow(2.0, n); i++) {
        printf("%.10lf\n", wp(i, n));
    }
}

int main() {
    cin >> n;
    for (int i = 0; i < (int)pow(2.0, n); i++) {
        cin >> ratings[i];
    }
    //solve();
    solve_by_loop();
    return 0;
}
```

参考

書籍

- プログラミングコンテスト攻略のためのアルゴリズムとデータ構造
- アルゴリズムイントロダクション

Web サイト

- Typical DP Contest (http://tdpc.contest.atcoder.jp/assignments)
- プログラミングコンテストでの動的計画法 (https://www.slideshare.net/iwiwi/ss-3578511)
- じじいのプログラミング (http://shindannin.hatenadiary.com/entry/20131208/1386512864)
- 競技プログラミングにおける区間DP問題まとめ (http://hamayanhamayan.hatenablog.jp/entry/2017/02/27/152226)