問題解決のためのプログラミング一巡り

金子 知適 (kaneko@acm.org)

2013 年夏学期

概 要

この資料は，東京大学教養学部前期課程で開講されている全学自由研究ゼミナール「実践的プログラミング」の 2013 年夏学期分の配布資料などを多少編集して，自習用資料として整形したものである．

プログラミングやアルゴリズムを学ぶためには，教科書を読むだけでなく実際に作ることが有用である．幸いにも，作成したプログラムの「正しさ」を自動で判定する，オンラインジャッジというシステムが国内外で充実し，優れた自習環境が整いつつある．この資料は，それらで学ぶ敷居をもう少し下げて，入門の段階の学習者が問題解決の面白さを味わう手助けを行う意図で準備された．たとえば，ほとんどの例題には，穴埋めや日本語で書かれた擬似コードを C++に翻訳するだけで完成するような，ヒントが用意されている．ヒントを追いながら手法の流れを理解したり，入出力データでプログラムの正しさをテストするなどの経験を積むことで，ヒントなしでもプログラムを組む力が身につくと期待している．この資料自体は様々な点で作成途上のものであるが，入門者向けの資料が現状で少ないため，活用の機会もあるかもしれないと考えて公開する．

目 次

第1章　はじめに

1.1　資料の構成

各章が 90 分の演習時間を想定して作られている．ヒントがついた易しい問題は，主に未経験者を想定して用意されていて，一旦理解した後であれば 5-10 分で解けるものが多い．しかし，初めて取り組む場合は時間を 5 倍程度長めに見積もることをお勧めする．経験者向けには，難易度の異なる複数の練習問題が紹介されている．所要時間は熟達度で異なるが，問題名に印 $\star$ が一つつくと，難易度が 5 倍程度 (たとえば回答作成に要する時間で測ったとして) 難化する目安である．また後ろの章の知識を前提としている場合もある．そのため，各章の問題を全て解いてから次に進むのではなく，印なしの易しい問題を解いたら一旦次の章に進むことを勧める．二週目には解ける問題が増えていることだろう．さらに印 $\star$ を二つ以上持つ問題は，その章の内容と多少は関係があっても解法と直接の関係がない場合もある．これは，どの戦略が有効かの見極めも，問題解決を学ぶ面白さの一つであるため．

教養学部前期課程実践的プログラミング履修 (予定) 者への補足　冬学期のセミナーが，この資料の予習を前提とすることは*ない*．すなわち，未経験者向けの例題と多少の解説，経験者向けの練習問題が引き続き提供される．扱うテーマは夏と重なる部分もあれば重ならない部分もある．

1.2　教科書・参考書

本文中で，参考書として「プログラミングコンテストチャレンジブック第二版」[1]の対応するページに言及することがある (言うまでもなくこの分野の名著である)．また，学習時間 (と予算) にゆとりがある者には，「アルゴリズムデザイン」[2]の 6 章までを時間をかけて読み進めることを勧める．ページ数が多いが，その分，丁寧に書かれているので類書の中では初学者に適すると思われる．(ただし筆者は英語版で読んだので，日本語版の評価は予想である)

1.3　オンラインジャッジシステム

本資料の問題は，以下のオンラインジャッジからとっている．資料作成時に担当者が各オンラインジャッジの利用条件を探した範囲では，この資料内での各問題への参照は問題ないと判断したが，お気づきの際は随時連絡されたい．

- Aizu Online Judge (AOJ) http://judge.u-aizu.ac.jp
- Peking University Judge Online for ACM/ICPC (POJ) http://poj.org
- Codeforces http://www.codeforces.com/
- MAIN.edu.pl http://main.edu.pl/en

[1] http://book.mycom.co.jp/book/978-4-8399-4106-2/978-4-8399-4106-2.shtml
[2] http://www.kyoritsu-pub.co.jp/bookdetail/9784320122178

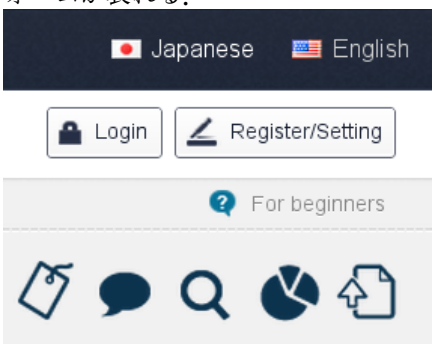
アカウント作成

初めての場合はまず AOJ のアカウントを作成する．各システムとも無料で使うことができる．なお，各オンラインジャッジは，運営者の好意で公開されているものであるから，迷惑をかけないように使うこと．特にパスワードを忘れないこと．

AOJ のアカウント作成 (初回のみ)　ページ右上の Register/Setting からアカウントを作成する．User ID と Password を覚えておくこと (ブラウザに覚えさせる，もしくは暗号化ファイルにメモする)．この通信は https でないので，注意．Affiliation は the University of Tokyo 等とする． E-mail や URL は記入不要．

ここで，自分が提出したプログラムを公開するかどうかを選ぶことができる．公開して ("public" を選択) いれば，プログラムの誤りを誰かから助けてもらう際に都合が良いかもしれない．一方，「他者のコード片の動作を試してみる」というようなことを行う場合は，著作権上の問題が発生しうるので，非公開の方が良いだろう ("private" を選択)．

AOJ への提出 (毎回)　ログイン後に問題文を表示した状態で，長方形に上向き矢印のアイコンを押すと，フォームが表れる．



自分の提出に対応する行 ("Author" を見よ) の "Status が "Accepted" なら正答．

正答でなかった時　様々な原因がありうるので，まずジャッジの応答がどれにあてはまるか，システムの使い方を誤解していないかなどを説明を読んで確認する．Submission notes (http://judge.u-aizu.ac.jp/onlinejudge/submission_note.jsp), Judge's replies (http://judge.u-aizu.ac.jp/onlinejudge/status_note.jsp), チュートリアル (http://judge.u-aizu.ac.jp/onlinejudge/AOJ_tutorial.pdf) などの資料がある．

一般的にプログラムが意図したとおりに動かないことは，誰でも (熟達者でも!) しばしばあることである．組み上げたプログラムが動かなかったとしても，何から何までダメという事はなく，多くの場合はほんの少しの変更で解決することが多い．そこで，どの部分までは正しく動いているか，各部品ごとに動作確認をする方針が有効である．コンピュータでのプログラミングは **copy** や **undo** ができることが長所であるから，(料理で食材を無駄にしてがっかりするようなことは起こらない)，臆せず色々試すと良い．

困った状況から復帰するノウハウも多少も存在する (付録 A) ので，徐々に身につけると有用であろう．一方で，経験が少ない段階では，15 分以上悩まないことをお勧めする．手掛かりなく悩んで時間を過ごすことは苦痛であるばかりでなく，初期の段階では学習効果もあまりないので，指導者や先輩，友達に頼る，あるいは一旦保留して他の問題に取り組んで経験を積む方が良いだろう．相談する場合は，「こう動くはずなのに (根拠はこう)，実際にはこう動く」と問題を具体化して言葉にしてゆくと解決が早い．なお，悩んで意味がある時間は，熟達に応じて 2 時間，2 日間等伸びるだろう．

1.4 実践例

1.4.1 ファイルの保存/コンパイル

各章では，サンプルや機能確認のために複数のコード片を扱う．そこで，混乱を避けるため，フォルダを作って，テーマごとに別の名前をつけてファイルに保存すると良い．そして，それぞれを動作可能に保つ．一方，お勧めしない方法は，一度作ったファイルを継ぎ足しながら，一つの巨大なファイルにすることである．そうしてしまうと，後から，2 種類のコードを実行して差を比べるようなことが難しくなる．

「ターミナル」(MacOSX の場合) の動作例は以下の通り:

```
% mkdir programming
（フォルダ作成，最初の一回のみ）
% mkdir programming/chapter1
（各章毎に行う）
% cd programming/chapter1
（カレントディレクトリを変更．$HOME/programming/chapter1/ 以下にソースコードを保存）
% g++ -Wall sample1.cc
（-Wall は警告を有効にするオプションで，何かメッセージが出た場合は解消することが望ましい．読み方が分からないメッセージが出た場合は，誰かと相談する）
% ./a.out
（実行）
```

1.4.2 標準入出力とリダイレクション

各問題では標準入出力を扱い，プログラムの正しさを入力に対する出力で判定する．入力では，入力データがある限り読み込んで処理する場合も多い．そこで，初めに例を挙げる．以下環境は基本的に MacOSX を想定するが，ubuntu や cygwin 等でも動作すると思われる．

例題

年を読み込んで，うるう年かどうかを判定し，日数を出力するプログラムを作る

(いい加減な) プログラム例 (C++): leap.cc

```
// 4 年に一度?
#include <iostream>
using namespace std;
int main() {
    int year;
    while (cin >> year) {
      if (year % 4 == 0)
        cout << 366 << endl;
      else
        cout << 365 << endl;
    }
}
```

C の場合: leap.c

```
/* 4 年に一度? */
#include <stdio.h>
int main() {
    int year;
    while (~scanf("%d", &year)) {
      if (year % 4 == 0)
        printf("%d\n", 366);
      else
        printf("%d\n", 365);
    }
}
```

1.4.3 コンパイル

(以降%記号は，ターミナルへのコマンド入力を示す)
C++の場合:

```
% g++ -Wall leap.cc
```

C の場合:

```
% gcc -Wall leap.c
```

1.4.4 実行例

実行例: 以下，斜体はキーボードからの入力を示す．終了は Ctrl キーを押しながら c または d をタイプする．この操作を^C や^D と表記する．

```
% ./a.out
2004
366
1999
365
1900
366
2000
366
^D
```

1.4.5 リダイレクションとファイルを用いたテスト

実行するたびに毎回キーボードをタイプするのは煩雑であるから，自動化したい．そこで，リダイレクションとファイルを用いたテストを解説する．早い段階で身につけることが望ましい．
正しい入出力例をエディタで作成し，cat コマンドで中身を確認する:

```
% cat years.input
2004
1999
```

```
1900
2000
% cat years.output
366
365
365
366
```

リダイレクションを使った実行 (キーボード入力の代わりにファイルから読み込む):

```
% ./a.out < years.input
366
365
366
366
```

実行結果をファイルに保存 (画面に表示する代わりにファイルに書き込む):

```
% ./a.out < years.input > test-output
% cat test-output
366
365
366
366
```

自動的な比較:

```
% diff -u test-output years.output
--- years.output        Fri Oct 14 10:53:52 2005
+++ test-output Fri Oct 14 10:53:56 2005
@@ -1,4 +1,4 @@
 366
 365
-365
+366
 366
```

4 行目あたりが違うことを教えてくれる

資料:

- HWB 15 コマンド
  http://hwb.ecc.u-tokyo.ac.jp/current/information/cui/
- HWB 14.4 コマンドを使ったファイル操作
  http://hwb.ecc.u-tokyo.ac.jp/current/information/filesystem/cui-fs/

第2章　配列, 二分探索

2.1　概要

様々な解法につながる基本として，配列から要素を探す問題と，その周辺をとりあげる．オンラインジャッジでは標準入出力を用いるので，その取り扱いに慣れる．また，プログラムを一度に作成するのではなく，部品ごとに作ってテストするスタイルを身につける．さらに，データの量に応じて，適切なアルゴリズムが必要になることを経験する．

2.2　言語機能: 配列, ループ, 条件分岐，関数

この章で使う言語機能を紹介する．これらの意味がわからない場合は，入門書を参照のこと．

2.2.1　配列, ループ, 条件分岐

```
int main() {
    int A[4] = { 0, 1, 2, 3, };
    for (int i=0; i<4; ++i)
        cout << A[i] << endl;
}
```

```
int main() {
    int A[4] = { 0, 1, 2, 3, };
    int i=0;
    while (i<4 && A[i]!=2)
        cout << A[i] << endl;
}
```

2.2.2　関数

```
double pi() { return 3.14; }
int main() {
   cout << 3.14 << endl;
   cout << pi() << endl;
}
```

```
int sum(int a, int b) { return a+b; }
int main() {
   cout << 3+5 << endl;
   cout << sum(3,5) << endl;
}
```

2.3 要素を探す

状況

ある人が家を片付けていて，手持ちの CD を友達に売りたい．うっかり二枚以上買ってしまったものもありうる (e.g., 重複あり)．買う人は，興味がある CD のリストを持っている (個人なので 2 枚以上はいらない, 重複なし)．何枚買ってもらえるか?

2.3.1 配列の要素が少ない場合

練習問題 Search I

整数の配列 S(売りたいリスト) と T(買いたいリスト) が与えられる．T に含まれる整数の中で S にも含まれる個数を出力せよ．

(入力の範囲)S と T の要素数はそれぞれ 100 以内．配列に含まれる要素は，0 以上 500 以下．

(注意)T の要素は互いに異なるが，S の要素は重複がある場合がある．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=10030

部品の作成: 入出力

標準入力からデータを読む (問題を解くために必要)．読んだ内容を標準出力に書き出す (入力方法に間違いがないかをテスト)．

```
#include <iostream>
using namespace std;
int SL, S[128], TL, T[128]; // 大域変数を利用，必要量より少し多めに
int main() {
    cin >> SL;
    for (int i=0; i<SL; ++i) cin >> S[i];
    cin >> TL;
    for (int i=0; i<TL; ++i) cin >> T[i];
    for (...) cout << S[i] << endl;
    for (...) cout << T[i] << endl;
}
```

(問題文では，配列 S と T の要素数が N と Q になっているが，対応が分かりにくいため，この資料では SL と TL とした．L は length の意図である．)

実行方法:

- `./a.out` とターミナルに入力し，キーボードから “Sample Input” のデータを入力する
- `./a.out` とターミナルに入力し，コピーペーストを用いて入力する
- 予め，“Sample Input” のデータを `sample-input.txt` などに保存しておく．その後，`./a.out < sample-input.txt` とリダイレクションを用いて実行．(推奨!)

入力した数値が出力されれば成功

部品の作成: 売られているか?

配列 S 内に同じ値があれば true, そうでなければ false を返す関数を作ろう.

```
int S[128];
bool is_member_of_S(int value) {
    for (S の全ての要素 S[j] について)
        if (S[j] が value と等しければ) return true;
    return false;
}
```

作成したら即座に，意図通りの動作をしているか確かめる.

```
int main() {
  S[0] = -3;
  if (is_member_of_S(-3)) cout << "found" << endl;
  else cout << "not found" << endl;
}
```

-3 以外の値や S[0] 以外の場所でも，適当に確かめる.

組立

これまでの道具を使って (新しいファイルを作って，今までのソースコードの必要な部分をコピーペーストすると良い)，問題に回答するプログラムを作成する．買取リスト (T) 内の要素一つ (CD 一枚づつ) について，売却リスト (S) 内に存在するかどうかを確認する．存在する枚数を数える．数を数える一つの方法は，合計を表す変数 (この場合は C) を 0 で初期化しておき，条件に合うものがあるたびに，1 増やすことである.

```
int main() {
  S, T の入力
  int C = 0;
  for (全ての T[i] について)
      if (is_member_of_S(T[i])) ++C; // S 内に T[i] がある
  cout << C << endl;
}
```

“Sample Input” を与えて，出力を確認せよ.

AOJ への提出

正しいと思うプログラムを提出してみよう．Accepted がもらえれば成功．実行結果の Time の欄が 00:00 sec かそれに近い数字であることを確認すること.

Wrong Answer 等であれば，何かが間違っているので原因を追求して修正する.

落穂ひろい (時間がなければ飛ばして良い)

確認: **T** と **S** の取替え　T と S を逆にするとプログラムの動作はどのように変わるだろうか.

```
int C = 0;
for (全ての S[j] について)
  if (is_member_of_T(S[j])) ++C;
cout << C << endl;
```

exp3-2-2.cc などとプログラムの名前を変えて保存し確かめてみよう．(1) “Sample Input” に対する出力は変わるだろうか． (2) 出力が変わる例を自作せよ

STL の利用　実は，`is_member_of_S(int value)` 相当の関数は，標準ライブラリに用意されているので，自分で書く必要はない．

```
#include <algorithm>
int a = count(S,S+SL,3); // S[0] から S[SL-1] までにある ‘‘3’’ の数を数える
```

関数 `count` の 3 つの引数は前から順に，探す範囲が `[S[0],S[SL-1]]` であること，探す要素が 3 であることを示す．参考: http://www.sgi.com/tech/stl/count.html

2.3.2　沢山の要素を探す

練習問題 Search II

Search I と同じ．ただし，入力に変更があり S の要素数は 10 万以内，T は 5 万以内．配列に含まれる要素も，10^7 まで大きくなりうる．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=10031

この大きさの変更には，どのような影響があるだろうか．まずは，Search I で書いたプログラムを配列の大きさを増やして，Search II に適用できるかどうか AOJ に提出してみよう．

入出力の変更 (*)　入出力がほぼそのまま使える．しかし，入力データが増えたので，配列の大きさを増やす必要がある．

```
int S[100100], T[50050];  // 配列の大きさを増やした
```

提出 (TLE)　結果は，Time Limit Exceeded となるはずである．この意味は，制限時間である 1 秒 (問題上部に “Time Limit : 1 sec” と記述がある) 以内に，プログラムの実行が終わらなかったという意味である．時間は，担当者の例では 3 分 41 秒となった．

解釈　何が遅いのか? Search I と Search II の差は S と T の大きさなので，これがどのように影響するかを考える．元のプログラムは，全ての S[i] と全ての T[j] の組み合わせについて一致を調べる．この組み合わせの種類は S と T の大きさに依存し，Search I ではそれぞれ 100 要素以内なので $100 \cdot 100 = 10^4$，Search II では，それぞれ 10 万要素と 5 万要素なので，$100,000 \cdot 50,000 = 5 \cdot 10^9$ となる．後者の方が 50 万倍なので，後者は前者の 50 万倍遅いと予想される．実際の計算時間は様々な要員に左右されるが，単純に 50 万倍であれば，前者が 0.001 秒だったとしても後者は 500 秒かかる見積もりとなる．

2.3.3　二分探索

そこで，関数 `is_member_of_S` を高速化して，制限時間内に解を出せるようにしよう．アイデアは以下の通り．

コンピュータの世界を離れて考えると，辞書や名簿は，番号順に並んでいることで効率的な検索が可能である．たとえば “Muller” という姓を 1024 ページの名簿から探す場合に，1024 ページの真ん中の 512 ページから始める．そのページが “M” より後なら，前を調べる．“M” 以前なら，後を調べる．いずれの場合でも，探す範囲を半分に絞ることが出来るといった具合である．

この探し方は，1 ページ目から順に 2,3 ページと最終ページまで名簿を探すより，速い．計算量のオーダを習った後であれば $O(n)$ と $O(log(n))$ であることを確認．

STL の利用　二分探索を初めて実装すると多少時間がかかるが，幸い C++の標準ライブラリには，二分探索の機能もソートの機能も含まれている．まずはそれらを利用してみよう．

```
#include <iostream>
#include <algorithm> // 追加

sort(S, S+SL); // 追加: 配列 S 内を昇順に並び替える
for (全ての T[i] について)
    if (binary_search(S, S+SL, T[i])) { // T[i] が S 内に存在する
       C = C+1;
    }
```

関数 sort と binary_search の 2 つの引数は，count 同様にソートする範囲や探す範囲が [S[0],S[SL-1]] であることを示す．sort(S, S+SL) のプラスの表記は，慣用的記法として理解すると良い．文法としては，配列は暗黙にポインタに変換されるので (&S[0], &S[0]+SL) と等価．(&S[0], &S[SL]) とも等価だが，S[SL] にアクセスすると配列の範囲を超えているので，普通はこのようには書かない．

参考: http://www.sgi.com/tech/stl/sort.html, http://www.sgi.com/tech/stl/binary_search.html

“Sample Input” で動作を確認した後，AOJ に提出してみよう．今度は，10 秒かからずに終了するはずである．

2.4　二分探索と周辺

配列 A[] から value を探す場合を多少形式的に書くと

1. 調べる区間を $[l, r)$ とする (範囲を表すには，left, right あるいは first, last などがしばしば用いられる)．0 ページ目から 1023 ページ目までを探す場合は，初期状態として l=0，r=1024 とする
2. l+n <= r となったら，探す範囲は最大で n ページしか残っていないので，[l,r) の範囲を順に探す．(典型的には n==1)
3. 区間の中央を求める m=(l+r)/2
4. value < A[m] なら (次は前半を探すので)r=m，そうでなければ (A[m] <= value すなわち A[m]==value の場合を含む) 後半を探すので l=m として 2 へ．(ここで m==l または m==r の場合は無限ループとなる．そうならないことを確認する．)

というような処理となる．

```
bool bsearch(const int array[], int first, int last, int value) {
    while (first + 1 < last) {
        int med = (first+last)/2;
        if (array[med] > value) last = med;
        else first = med;
    }
    return first < last && array[first] == value;
}
```

自分で二分探索実装して，Search II を解いてみよう

2.5　応用問題

2.5.1　最小値の最大化: Aggressive Cows

配列から値を探す状況以外にも，二分探索の考え方を用いると綺麗に解ける問題もある．

練習問題 **Aggressive Cows**⋆

直線上に N 棟の牛舎 (個室) がある．C 頭の牛を，なるべく互いを離して入れたい．

http://poj.org/problem?id=2456

```
bool ok(int M) {
  距離 M 以上離して配置できれば true，そうでなければ false
}
int main() {
    問題読み込み
    int l = 0, h = 大きな数;
    while (l+1 < h) {
        int m = (l+h)/2;
        if (ok(m)) l = m; else h = m;
    }
    printf("%d\n", l);
}
```

あとから

2.5.2 答えのコストが異なる場合: The Search

二分探索は，区間を半分に分ける (以上/未満のどちらの場合でも，その後の作業の (最悪) コストが等しい) ことで効率を上げている．以上/未満のどちらかで，コストが異なる場合は最適な戦略が異なる．以下の問題では，a=b の時に二分探索が最適解となる．(この問題を解くには，動的計画法 (8 章) を必要とする)

練習問題 **The Search (16th Polish Olympiad in Informatics)**⋆⋆

お姫様が幽閉されている階を当てる．“n 階より上/下か?” という問に答えてくれる．答えの真偽に応じて，a 円または b 円取られる．(最悪ケースの) 支出を最小化する質問戦略を求めよ．

http://main.edu.pl/en/archive/oi/16/pos

第3章　計算時間の見積りと全通りの検査

3.1　概要

基本演算の回数

コンピュータが得意な解法は全部を力づくで試すことである．実際に多くの問題をそれで解くことが出来る．
ただし問題が大きくなると，実行時間が増える．多くのコンテストの問題では実行時間に制限がついているし，実用に用いるプログラムでも，何らかの実行速度に関する要請がある場合が多い．プログラムを書き終えてから速度に問題があることが分かると，書き直しが困難な場合もあるので，プログラムを書く*前*に何らかの見通しを得ておくことが望ましい．
現実の計算機は複雑な装置であるから，単純なモデルに基づく目安を考える．たとえば，プログラムを実行する過程で，加減乗除のような基本演算が何回行われるかを数える．このモデルでは加算と除算では速度が異なるとか，同時に二つの命令を実行される場合があるなどの，細かい点は無視している．目安であるので現実との対応関係は別に議論が必要だが，役に立つ場面も多い．

表 3.1: 1 秒間に実行可能な計算単位 (参考書 p.20)

1,000,000	間に合う
10,000,000	たぶん間に合う
100,000,000	間に合わなくてもやむを得ない

3.2　next_permutation による全通りの検査

練習問題 **Rummy (UTPC2008)**

9 枚のカードを使うゲーム．手札が勝利状態になっているかどうかを判定してほしい．勝利状態は，手札が 3 枚ずつ 3 つの「セット」になっていること． セットとは，同じ色の 3 枚のカードからなる組で，同じ数 (1,1,1 など) または連番 (1,2,3 など) をなしているもの．

カードは，赤緑青の三色で，数字は 1-9 まで．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2102&lang=jp

3.2.1 入力例と解釈

```
1 1 1 3 4 5 6 6 6
R R B G G G R R R
```

3,4,5 と 6,6,6 はセットだが 1 は色が違う ➔ 勝利状態ではない

```
2 2 2 3 3 3 1 1 1
R G B R G B R G B
```

1,1,1 等同じ数で揃えようとするセットにならないが，1,2,3 の連番を同色で揃えられる ➔ 勝利状態

3.2.2 解答作成

方針

人間が役に当てはまるかを検討する場合，賢い (比較的複雑な) 試行を行うことで比較的少ない試行錯誤で最終的な判断にいたる，と想像される.「人間がどう考えるか」をコンピュータ上で実現することは人工知能の興味深い目標となることが多いが，コンピュータにとって最も簡単な方法は別に存在することも多い. (例: 122334 という並びを見て，慣れた人間は一目で 123, 234 と分解できるが，この操作を例外のない厳密なルールにできるだろうか?)

ここでは，(1) カードの全ての並び替えを列挙する (2) 各並び順につき，前から 3 枚づつセットになっていることを確かめる，という方針で作成しよう．このような,「単純な試行で全ての可能性を試す」アプローチはコンピュータで問題解決を行う時に適していることが多い．ここで問題になるのは,「全ての可能性」の場合の数である．この場合にカードの並び替えの種類は，9 の階乗 (=36 万超) であるから，余裕を持って処理可能である．仮にカードの数が 15 枚であると場合の数は 1 兆を越え，制限時間の 5 秒以内に答えを出すことは絶望的となる．演習で出題される問題ではなく実用の問題に取り組む場合には，このような見積もりが特に重要である．

3.2.3 permutation の列挙

まず，配列の中身を並べ替える道具を作ってみよう．幸い C++の標準ライブラリには `next_permutation` という関数があるので，これを使うと便利である．もしこの関数を自作する場合には，間違える箇所が多いと予想されるので，入念にテストを行うこと．典型的な使い方は以下のように `do .. while` ループと組み合わせて，全ての並び替えを昇順に列挙することである．

```
#include <algorithm>
    int A[4] = {1,1,2,3}; // 昇順に並べておく
    do {
        cout << A[0] << A[1] << A[2] << A[3] << endl;
    } while (next_permutation(A,A+4));
```

出力を確認せよ (実際に場合が尽くされているか?):

```
1123
1132
...
3121
3211
```

`next_permutation` は，まだ試していない組み合わせが (正確には全ての組み合わせを昇順に並べた時の次の要素が) ある場合，配列の中身を並び替えて真を返す．そうでない場合は偽を返す．`next_permutation` が偽を返すとループが終了する．

配列の要素が昇順に並んでいない状態に初期化して (たとえば `int A[4] = {2,1,1,3};`)，上記のコードを実行すると，出力はどのように変わるか．

入力の処理

例によって，入力をそのまま出力することから始める．card[i] $(0 \leq i \leq 8)$ が i 番目のカードを表すことにする．

```
#include <iostream>
#include <string>
using namespace std;
int T, card[16];
int main() {
    cin >> T;
    for (int t=0; t<T; ++t) {
        for (int i=0; i<9; ++i) {
          cin >> card[i];
          card[i] を出力
        }
        string color;
        for (int i=0; i<9; ++i) {
          cin >> color;
          color を出力
        }
    }
}
```

色の変換

入力を処理した段階で，各カードは $\langle$ 色, 数 $\rangle$ という二つの情報の組み合わせからなっているが，これを整数で表現すると今後の操作で都合が良い．そこで，赤の [1,9] のカードをそれぞれ [1,9] で，緑の [1,9] のカードをそれぞれ [11,19], 青の [1,9] のカードをそれぞれ [21,29] で表現することにしよう．(例 G3 を 13 で表し，B9 を 29 で表す) [1]

```
    cin >> T;
    for (int t=0; t<T; ++t) {
        for (int i=0; i<9; ++i) {
          cin >> card[i];
        }
        string color;
        for (int i=0; i<9; ++i) {
          cin >> color;
            if (color == "G") card[i] += 10;
            else if (color == "B") card[i] += 20;
          card[i] を出力して確認
        }
    }
```

セットの判定

続いて，3 枚のカードがセットになっているかどうかを判定する．問題にあるようにセットになる条件は二つある．それぞれを別に作成しテストする．

```
bool is_good_set(int a, int b, int c) {
  return is_same_number(a, b, c) || is_sequence(a, b, c);
}
```

[1] この変換は色の異なる全てのカードが異なる数値になれば良いので青を [100,109] 等であらわしても構わない．

一つの条件は，同じ色かつ同じ数値の場合である．card[i] においては，異なる色は異なる整数で表現されているので，単に数値が同じかどうかを調べれば良い．

```
bool is_same_number(int a, int b, int c) {
  a と b と c が同じなら真，それ以外は偽
}
```

もう一つは，同じ色かつ同じ数値の場合である．card[i] においては，異なる色は異なる整数で表現されていて，かつ，0 や 10 という数値は存在しないので，単に数値が連番かどうかを調べれば良い．

```
bool is_sequence(int a, int b, int c) {
  a+2 と b+1 と c が等しければ真，それ以外は偽
}
```

練習: なぜ「a-2 と b-1 と c が等しい」という降順の連番を考慮する必要がないのか考察しなさい．(後回し可)

テスト例:

```
int main() {
  // is_same_number のテスト
  cout << is_same_number(3, 4, 5) << endl; // 偽
  cout << is_same_number(3, 3, 3) << endl; // 真
  // is_sequence のテスト
  cout << is_sequence(3, 4, 5) << endl; // 真
  cout << is_sequence(3, 3, 3) << endl; // 偽
  // is_good_set のテスト
  cout << is_good_set(3, 4, 5) << endl;  // 真
  cout << is_good_set(3, 3, 3) << endl;  // 真
  cout << is_good_set(3, 3, 30) << endl; // 偽
  cout << is_good_set(5, 4, 3) << endl;  // 偽
}
```

勝利状態の判定

global 変数 card が勝利状態にあるかどうかを判定する

```
bool is_all_good_set() {
    return ((card[0],card[1],card[2] が good set)
           かつ (card[3],card[4],card[5] が good set)
           かつ (card[6],card[7],card[8] が good set));
}
```

全体の組み立て

全ての並び順を作るコードと is_all_good_set() を組み合わせると，ほぼ完成である．

```
int win() {
    card をソートする
    do {
       この card の並び順が勝利状態なら return 1;
    } while (next_permutation(card, card+9));
    // 全ての組み合わせを試したが勝利状態にならなかった
    return 0;
}
```

各データセットを読み込み後にこの関数 win() を呼び，その返り値の 1 または 0 を出力するプログラムを作成せよ．手元でテスト後に，AOJ に提出して accepted になることを確認せよ．

3.3 試す種類を減らす

練習問題 Space Coconut Grab (模擬国内予選 2007)

宇宙ヤシガニの出現場所を探す．エネルギー E が観測されたとして，出現場所の候補は，$x+y^2+z^3=E$ を満たす整数座標 (x,y,z) で，さらに x+y+z の値が最小の場所に限られるという．x+y+z の最小値を求めよ．(x,y,z は非負の整数，E は正の整数で 1,000,000 以下，宇宙ヤシガニは，宇宙最大とされる甲殻類であり，成長後の体長は 400 メートル以上，足を広げれば 1,000 メートル以上)

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2012

まず，全部試して間に合うかを考える．(間に合うならそれが一番実装が簡単なプログラムである)

変数 x,y,z の動く範囲を考えると，それぞれ x:[0,E], y:[0,$\sqrt{E}$], z:[0,$\sqrt[3]{E}$] である．それら (x,y,z) の組み合わせは，E の最大値は 1,000,000= 10^6 であるから，最大 $10^6 \cdot 10^3 \cdot 10^2 = 10^{11}$ である．一方，表 3.1 の最終行に記載された，ギリギリ間に合うかもしれないという値が 1 億であるから，制限時間が 8 秒であることを加味しても，この方針では間に合いそうにない．

減らす指針として，全ての (x,y,z) の組み合わせを考える必要はなく，$x+y^2+z^3=E$ を満たす範囲だけで良いことを用いる．たとえば，x と y を決めると，E から z は計算できる (z が整数とならない場合は無視して良い)．この方針で調べる種類は，(x,y) の組み合わせの種類である，$10^6 \cdot 10^3$ となる．この値はまだ大きいが，先ほどと比べると 1/100 に削減できている．上記と同様に，x と z を決めて y を求める，z,y を決めて x を求めることもできる．それらの方針の場合に，調べる組み合わせの種類を求めよ．一番少ないものが表 3.1 で間に合う範囲であることを確認し，実際に実装して AOJ に提出して確かめる．

3.4 効率良く調べる

練習問題 Pilot (17th Polish Olympiad in Informatics)⋆

パイロットがどの程度まっすぐ飛べるかどうかを計算する．時刻毎の位置 (一次元) が数列 a_i として与えられる．数列の長さは最大 3,000,000 である．ずれてよい範囲として t が与えられる．数列の範囲 $[i,j]$ のうち，範囲内の全ての k,l について $|a_k-a_l| \le t$ という条件をみたす最長の範囲を求めよ．(詳しくは原文参照)

http://main.edu.pl/en/archive/oi/17/pil

考え方: i の候補は 300 万通り，j の候補も同じだけある．$i \le j$ という制約をつけても，i と j の候補を全て調べることは，表 3.1 に照らして無理である．

参考: 値が大きいので注意する．入力も多いので cout だと間に合わず (70 点くらい)，scanf を使う．$O(n \log n)$ の回答だと 70 点くらい．

第4章　繰り返し二乗法と行列の冪乗

4.1　概要

適切な手法を用いると，計算時間を短縮できることを体験する．この手法が適用可能な問題では，10 億ステップ後のシミュレーション結果を簡単に求めることもできる．

繰り返し二乗法

応用先:

- すごろくでサイコロで $N(0 \leq N \leq 2^{31})$ が出た時に，行ける場所を全て求める
- 規則に従って繁殖する生物コロニーの N ターン後の状態を求める
- 規則に従った塗り分けが何通りあるかを求める

(参考書 p.114)
例: $3^8 = 6561$ の計算

- $3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$ ➔ 7 回の乗算
- `int a = 3*3, b = a*a, c = b*b;` ➔ 3 回の乗算

練習: 3^{128} の場合は? ➔ $\log(128)$ 回の乗算

関連:

- オーバーフローに注意: int で表せる範囲は，この環境では約 20 億くらい
- 剰余の扱い: (a*b)%M = ((a%M)*(b%M))%M

4.2　言語機能: struct と再帰関数

この章では `long long` と `struct` を使う．`struct` の書式に不安がある場合は，入門書を復習のこと．再帰関数は，この段階では場合によってはとばしても良い．

4.2.1　long long

本資料が前提とする環境で，GCC, G++で 64bit 整数を用いる場合は，long long という型を用いる．

```
long long a = 1000000;
int b = 1000000;
cout << a * a << endl; // 1000000000000
cout << b * b << endl; // -727379968 (オーバーフロー)
```

4.2.2 struct

```
struct Student {
  int height, weight;
};
```

使用例:

```
  Student a;
  a.height = 150;
  a.weight = 50;
  cout << a.height << ' ' << a.weight << endl;

  Student b = { 170, 70 };
  cout << b.height << ' ' << b.weight << endl;

  Student c = { 180 }; // weight == 0
  Student d = { };  // height, weight == 0
```

4.2.3 再帰関数

```
int factorial(int n) { // 階乗
    if (n <= 1) return 1;
    return n*factorial(n-1); // 自分を呼び出す
}
```

使用例:

```
  cout << factorial(5) << endl;
```

4.3 正方行列の表現と演算

メモ:以下に 2x2 の行列のサンプルコードを掲載する．配列や vector, valarray 等のデータ構造を用いると NxN の行列を自作することもできる．研究や仕事で必要な場合は，専用のライブラリを使う方が無難．

```
struct Matrix2x2 {
  int a, b, c, d; // a,b が上の行，c,d が下の行とする
};
```

表示も作っておこう

```
void show(Matrix2x2 A) {
    cout << "[ " << endl
         << A.a << ' ' << A.b << endl
         << A.c << ' ' << A.d << endl
         << "]" << endl;
}
```

行列同士の乗算　続いて乗算を定義する．以下の mult は行列 A，B の積を計算する．

```
// returns C = A*B
Matrix2x2 mult(Matrix2x2 A, Matrix2x2 B) {
    Matrix2x2 C = {0}; // 0 で初期化
    C.a = A.a * B.a + A.b * B.c;
    C.b = A.a * B.b + A.b * B.d;
    C.c = A.c * B.a + A.d * B.c;
    C.d = A.c * B.b + A.d * B.d;
    return C;
}
// (注) 冪乗計算はすぐにオーバーフローするので注意：ここに細工をすることも多い
```

使用例:

```
  Matrix2x2 A = {0,1, 2,3}, B = {0, 1, 2, 0};
  show(A);
  show(B);
  Matrix2x2 C = mult(A, B);
  show(C);
```

冪乗の計算 (繰り返し自乗法)　次のコードは行列 A の $p(>0)$ 乗を計算し, O に書きこむ.

```
// O = A^p
Matrix2x2 expt(Matrix2x2 A, int p) {
    if (p == 1) {
        return A;
    } else if (p % 2) {
        Matrix2x2 T = expt(A, p-1);
        return mult(A, T);
    } else {
        Matrix2x2 T = expt(A, p/2);
        return mult(T, T);
    }
}
```

使用例:

```
  Matrix2x2 A = {0,1, 2,3};
  Matrix2x2 C = expt(A, 3); // A の 3 乗
  show(C);
```

4.4　練習: フィボナッチ数

練習問題 Fibonacci (Stanford Local 2006)

フィボナッチ数列の，n 項目の値を 10^4 で割った余りを計算しなさい.

$0 \leq n \leq 10^{16}$

http://poj.org/problem?id=3070

4.4.1　様々な計算方針

方針 1: 定義通りに計算する

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-2)+fib(n-1);
}
```

使用例:

```
int main() {
  for (int i=1; i<1000; ++i)
    cout << "fib" << i << " = " << fib(i) << '\n';
}
```

n = 30 くらいで，先に進まなくなる.

方針 **2:** 一度行なった計算を記憶する

table という配列を用意し，一度行なった計算を記憶させてみよう．この工夫は，メモ化 (**memoization**, **tabling**) などと呼ばれる，応用範囲の広いテクニックである．

```
int table[2000]; // 2000 まで答えを記憶
int fibmemo(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (table[n] == 0) // もし初見なら
      table[n] = fibmemo(n-2)+fibmemo(n-1); // 計算して覚える
    return table[n]; // 覚えていた値を返す
}
```

こんどは，n=1000 でもすぐに答えを得られる．(オーバーフローしているが，ここでは一旦無視する)

ただし，この方法では 配列 `table` の要素数までしか計算することができない．問題では，n の最大値は 10^{16} なので，現実的でない．

方針 **3:** 小さい方から計算する

n に比例する時間で計算可能である．この方法では，10^9 くらいなら待っていれば終わるが，10^{16} は時間がかかりすぎる．

```
int fibl(int n) {
    int a[2] = {0,1};
    for (int i=2; i<=n; ++i) {
        // 不変条件: ループ開始時に，a[0] と a[1] はそれぞれ
        //   Fib(i-1) と Fib(i-2) (i が奇数)
        //   Fib(i-2) と Fib(i-1) (i が偶数)
        // に相当
        a[i%2] = (a[0]+a[1])%10000;
    }
    return a[n%2];
}
```

4.4.2 行列で表現する

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

$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ とすると，結合則から以下を得る:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = A^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

この方針であれば，n が 10^{16} まで大きくなっても現実的に計算できる．ただし，n が `int` で表せる範囲を超えるので，`expt()` の引数などでは `long long` を用いること．また，普通に計算すると要素の値も `int` で表せる範囲を越えるので，

- (a*b)%M = ((a%M)*(b%M))%M
- (a+b)%M = ((a%M)+(b%M))%M

などの性質を用いて，小さな範囲に保つ．

動作確認には，小さな n に対して，方針 3 の手法などと比較して答えが一致することを確認すると良い．10^4 の剰余は，$F_{10} = 55$, $F_{30} = 2040$ などとなる．

4.5 応用問題

練習問題 One-Dimensional Cellular Automaton (アジア地区予選東京 2012)

(意訳) 行列を T 乗する $(0 \leq T \leq 10^9)$

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1327

NxN の行列の実装例:

```
struct Matrix {
    valarray<int> a;
    Matrix() : a(N*N)  a=0;
};
valarray<int> S;
Matrix multiply(const Matrix& A, const Matrix& B) {
    Matrix C;
    for (int i=0; i<N; ++i)
        for (int j=0; j<N; ++j)
            C.a[i*N+j] = (A.a[slice(i*N,N,1)]*B.a[slice(j,N,N)]).sum()%M;
    return C;
}
```

練習問題 行けるかな? (UTPC 2008)⋆

サイコロが巨大なすごろく (目の合計が 2^{31} まで)

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2107

- U ターン禁止の表現: 有向辺 (道+向き) に番号を振り，ある番号 (が表す辺) から次にどの番号 (があらわす辺) に行けるかを表す，遷移行列を作る
- n ターン後に行ける場所には，n 乗した遷移行列と初期位置を表すベクトルの積が対応する．
- 隣接行列などのグラフの表現は，9 章に目を通した後に戻ってくることを推奨．

練習問題 Numbers (GCJ 2008 Round1A C)⋆

概要: $(3+\sqrt{5})^n$ の最後の 3 桁を求める

http://code.google.com/codejam/contest/32016/dashboard#s=p2

(参考書 p.239)

GCJ への提出方法

- プログラムを作る: Sample で確認する
- “Solve C-small” をクリックして C-small-practice.in をダウンロードする

- `./a.out < C-small-practice.in > output.txt` のようにリダイレクションで，出力を作る
  豆知識: `./a.out < C-small-practice.in | tee output.txt` のように `tee` というコマンドを使うと，ファイルに保存しながら端末に出力してくれるので，進行状況が分かる．
- “Submit file” から `output.txt` を提出する．(即座に判定が表示される)
- 正解したら， “Solve C-large” から，large も解く

練習問題 Leonard Numbers (POI Training Camps 2008)⋆⋆

Fibonacci 数に似た，Leonard 数の和を求めよ

http://main.edu.pl/en/archive/ontak/2008/leo

第5章　平面の幾何 (1)

5.1　概要: 点の表現と演算

概要

幾何の問題を扱う基本を紹介する (参考書 p. 229). 計算機で図形を扱う場合には，人間が目で見ると簡単な概念でも，プログラムで記述すると思ったより難しい場合もある．また浮動小数 (double x, y など) を扱うので，誤差に注意する必要もある．並行や図形の内外などよく馴染んだ概念を,「符号付き三角形の面積」という道具で，表してみよう.

C++の場合は，複素数で表現すると少し楽 (実部 real() が x で，虚部を imag()y に対応させる):

```
#include <complex>
#include <cmath>
typedef complex<double> xy_t;
xy_t P(1, -2), Q; // 初期化
cout << P << endl; // (debug 用) 表示
cout << P.real() << endl; // x 座標
cout << P.imag() << endl; // y 座標
P += xy_t(3, 4); // 平行移動
P *= xy_t(cos(a), sin(a)); // 回転
cout << abs(P) << endl; // 長さ
```

C (gcc) の場合

```
#include <complex.h>
#include <math.h>
complex a = 0.0 + 1.0I; // 初期化
complex b = cos(3.14/4) + sin(3.14/4)*I;
printf("%f %f\n", creal(a), cimag(a)); // 実部，虚部
a *= b; // 掛け算
printf("%f %f\n", creal(a), cimag(a));
```

よく使う演算

```
// 内積: a.x*b.x +a.y*b.y
double dot_product(xy_t a, xy_t b) { return (conj(a)*b).real(); }
// 外積，ベクトル a,b が作る三角形の符号付き面積の二倍: a.x*b.y - b.x*a.y
double cross_product(xy_t a, xy_t b) { return (conj(a)*b).imag(); }
// 投影
xy_t projection(xy_t p, xy_t b) { return b*dot_product(p,b)/norm(b); }
```

5.2 三角形の符号付き面積の利用

5.2.1 平行の判定

練習問題 Parallelism (PC 甲子園 2003)

概要: A = (x1, y1), B = (x2, y2), C = (x3, y3), D = (x4, y4) の異なる 4 つの座標点が与えられたとき，直線 AB と CD が平行かどうかを判定せよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0021&lang=jp

回答方針: ベクトル AB とベクトル CD からなる三角形が面積を持つかどうかを判定すれば良い

```
const double eps = 1e-11;
double x[4], y[4];
int N;
int main() {
    cin >> N; // 問題数
    for (int t=0; t<N; ++t) {
        for (int i=0; i<4; ++i)
            cin >> x[i] >> y[i]; // x0,y0..x3,y3
        xy_t a[2] = {
            xy_t(x[0],y[0]) - xy_t(x[1],y[1]),
            xy_t(x[2],y[2]) - xy_t(x[3],y[3])
        };
        bool p = abs(a[0] と a[1] の符号付き面積) < eps;
        cout << (p ? "YES" : "NO") << endl;
    }
}
```

数値誤差の取り扱い ⋆ double などの浮動小数を用いる時には，$\frac{1}{2}$ の冪乗の和で表される数値以外は，必然的に誤差を含む．この問題での入力は，絶対値が 100 以下かつ各値は小数点以下最大 5 桁までの数値と明示されたので，10^5 倍して整数 (long long) で扱えば誤差の影響を避けることができる．あるいは，サンプルコードの eps のように，誤差の範囲を予測する方法もある．二つのベクトル (a, b) と (c, d) にそれぞれの要素に誤差が加わった時に，(1) 平行の場合に $|ad - bc|$ の取る最大値 (誤差がなければ 0) と，(2) 平行でない場合に $|ad - bc|$ の取る最小値を比較して，(1)< 閾値 <(2) となるよう閾値をとる．粗く見積もると (1) は最大 $(4 \cdot 100) \cdot (100 \cdot 2^{-54}) \approx 2.2 \cdot 10^{-12}$ 程度 ($100 \cdot 2^{-54}$ は 100 までの数を double で表した時の表現誤差，400 は $|(a+\epsilon)(d+\epsilon) - (b+\epsilon)(c+\epsilon)|$ を展開した時の ϵ にかかる係数の見積もり), (2) は 10^{-10} 程度 (入力が表現可能な 10^{-5} 値の自乗より).

5.2.2 内外判定

練習問題 A Point in a Triangle (PC 甲子園 2003)

平面上に (x1, y1), (x2, y2), (x3, y3) を頂点とした三角形と点 P(xp, yp) がある．点 P が三角形の内部 (三角形の頂点や辺上は含まない) にあるかどうかを判定せよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0012&lang=jp

回答例: 三角形の各点を a,b,c とすると，３つの三角形 pab, pbc, pca の符号付き面積を考える．p が abc の内部にあれば符号は一致し，外部にあれば一致しない．

```
double x[4], y[4];
int main() {
    while (true) {
        for (int i=0; i<4; ++i) cin >> x[i] >> y[i];
        if (!cin) break;
        xy_t a(x[0],y[0]), b(x[1],y[1]), c(x[2],y[2]), p(x[3],y[3]);
        // pab の符号付き面積の 2 倍は，cross_product(a-p,b-p)
        // pbc の符号付き面積の 2 倍は，cross_product(b-p,c-p)
        // pca の符号付き面積の 2 倍は，cross_product(c-p,a-p)
        bool ok = 符号が揃っている
        cout << (ok ? "YES" : "NO") << endl;
    }
}
```

5.3 応用問題

練習問題 Circle and Points (国内予選 2004)⋆

xy 平面上に N 個の点が与えられる．半径 1 の円を xy 平面 上で動かして，それらの点をなるべくたくさん囲むようにする．このとき，最大でいくつの点を同時に囲めるかを答えなさい．ここで，ある円が点を「囲む」 とは，その点が円の内部または円周上にあるときをいう． (この問題文には誤差に関する十分な記述がある)

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1132&lang=jp

候補となる円の位置は無数にあるので，候補を絞る．(「ぎりぎりを考えよ」参考書 p.229)

同時に囲める最大の点を n として，n 個を囲んだ円があったとする．もしその円に点が接してないのであれば，内部の点のどれかが接するまで動かしても，囲んでいる点の数は変わらない．すなわち，点のどれかと接する円だけを考えれば十分である (残りの円を考慮に入れても，答えは変わらない)．しかしそのような円はまだ無数にある．

n 個を囲んだ円があり，一点が円上に乗っているとする．その点を中心に円を回転させると，内部の点が新たに円と接するまで動かしても，囲んでいる点の数はかわらない．すなわち，2 点を通る円だけを考えれば十分である (残りの円を考慮に入れても，答えは変わらない)．そのような $2N^2$ 程度しかない．

例外: 答えが 1 の時

回答例:

```
int main() {
  最大値=1
  for (点 p) {
    for (点 q) { // p!=q
      if (pq を通る円があれば) { // 0 個,1 個,2 個の場合がある
          全ての点を確かめながら，内部の個数を数える
          最大値を越えていれば更新する
      }
    }
  }
}
```

練習問題 Altars (6th Polish Olympiad in Informatics)⋆⋆

中国では悪霊は直線上に進むと信じられているという枕．

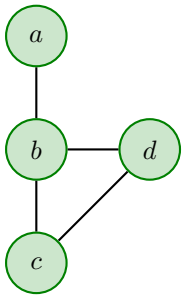
長方形の寺院があり，中央に祭壇がある．寺院の壁は，東西または南北のいずれかの方向からなる．寺院の入り口は，ある辺の中央にある．外部から祭壇に視線が通っているかどうかを調べよ．

http://main.edu.pl/en/archive/oi/6/olt

第6章　グラフと木

6.1　概要: グラフ

接続関係に焦点をあてて世の中をモデル化する際に，グラフがしばしば用いられる．路線図，物流，血縁関係，などなど．(参考書 2-5 あれもこれも実は “グラフ” p. 87)



用語

グラフは，頂点 (点, 節点,vertex; 複数形は vertices) と辺 (枝，線，edge, arc) からなる．頂点集合 V と辺集合 E でグラフ $G = (V, E)$ を表す．

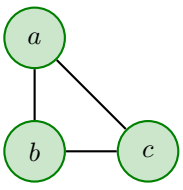
例: 3 つの頂点 $V = \{1, 2, 3\}$ の辺を全て結んだグラフ (=三角形) は，$E = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$

頂点 v と辺 e に対して $v \in e$ となる時，v は e に接続する．頂点 v にたいして，v の接続辺の個数 $|E(v)|$ を次数 $d(v)$ という．二つの頂点が共通の接続辺を持つ場合にその頂点は隣接するという．隣接する頂点をリストにして並べたものをパス (path, trail, walk で細かくは異なる意味を持たせるので，詳しく学ぶ際には注意) と呼ぶことにする．

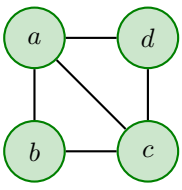
例: 三角形の各頂点の次数は 2

グラフの辺をちょうど一回づつ通る閉路/パスを **Euler circuit/trail (オイラー閉路/路)** という．

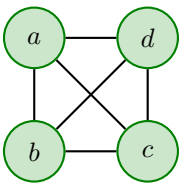
判定法: 連結であること，全ての頂点の次数が偶数であること/パスの始点と終点を除いて偶数であること．



Euler 閉路あり (e.g., abca)



閉路はないが全ての辺を辿れる (adcabc)



全ての辺をたどれない

6.2 一筆書きの判定

練習問題 Patrol (PC 甲子園 2005)

問題: パトロールする街路を一筆書きで辿れるかを判定せよ．始点と終点は指定されている．連結であることは保証されている．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0086

回答例

```
int main() {
    while (辺 a,b を読み込む) {
        頂点 a の次数を増やす;
        頂点 b の次数を増やす;
        if (a == 0) {
            始点から終点に一筆書き出来るか (*) を判定して出力
                (*) 頂点番号 1 と 2 の字数が奇数，かつ，
                    頂点番号 3 以上の頂点の次数が全て偶数
            次のテストケースのために，全ての頂点の次数を 0 に戻す
        }
    }
```

練習問題 Play on Words (Central Europe 1999)⋆

単語が与えられるので，全ての単語を「しりとり」で繋げられるかを判定せよ．初めと終わりの単語は自由に選んで良い．(連結であることは保証されていない)

筆者注: 時間制限が厳しいので入力には scanf を使うと良い

http://poj.org/problem?id=1386

回答例　アルファベットを頂点としたグラフを考える．たとえば news という単語を，頂点 n から s への辺と考える．ここで反対向きには使えないことから，(先ほどと異なり) 辺に向きがある (有向グラフという)．

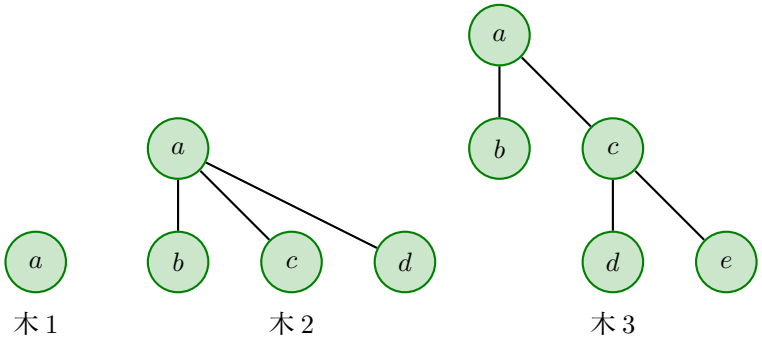
有向グラフの次数は，辺の向きに対応して，入次数 (in-degree) と出次数 (out-degree) を区別して扱う．

有向グラフがオイラー閉路を持つ必要十分条件は，連結であることと，全ての頂点の入次数と出次数が等しいことである．閉路でない場合は，パスの始点/終点で入次数が出次数より一つ多い/少ない．

問題 A と異なり，連結性が保証されないので，自分で判定する．(9 章参照)

6.3 木

特殊な (連結で閉路がない) グラフを木という．木は，一般のグラフより扱いやすい．木で表せるものには，式 (3 + (5 - 2))，階層ファイルシステム (リンクなどを除く)，自分の祖先を表す家系図 (いとこなど血縁間の結婚がない場合)，インターネットのドメイン名などがある．



定義

無向グラフ G に対して，全ての 2 頂点 v, w に足して $v-w$ パスが存在する時に G を連結と呼ぶ．閉路を含まないグラフを森 **(forest)** または林と呼ぶ．連結な森を木 **(tree)** という．

グラフ T の頂点数が n として，T が木であることと以下は同値である:

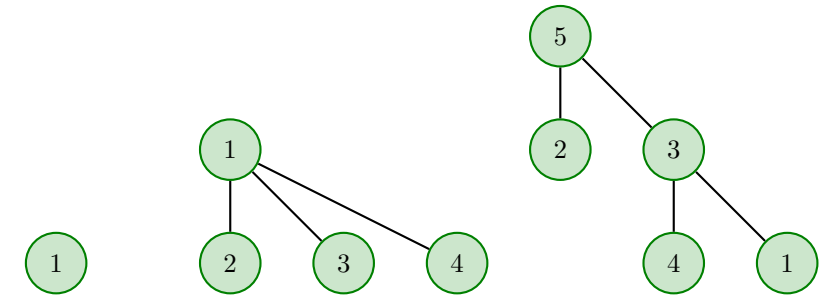
- T は $n-1$ 個の辺からなり，閉路を持たない
- T は $n-1$ 個の辺からなり，連結である
- T の任意の 2 頂点に対して，2 頂点を結ぶパスが一つのみ存在する
- T は連結で，T のどの辺についても，それを T から取りのぞいたグラフは非連結
- T は閉路を含まず，T の隣接しない 2 頂点を xy をどのように選んでも，xy をつなぐ辺を T に加えたグラフは閉路を含む

木の頂点の特別な一つを根 **(root)** と呼ぶ．グラフを図示する際には，根を一番上または下に配置することが多い．次数 1 の点を葉 **(leaf)** と呼ぶ．

6.4 「親」を使った木の表現

N 個の節点を持つ「木」を計算機上で表す方法の一つに，各節点に 1 から N までの数字の番号をふり，各節点の親を一次元配列 (以下，parent の頭文字を用いて P[] と表記する) で管理する方法がある．木には，たかだか 1 つの親を持つという性質がある．そこで，節点 i に対応する $P[i]$ の数値に，節点 i が親を持つ場合にはには親の番号，親を持たない場合は -1 または自分自身などの特殊な番号を割り当てる．

なお，各節点で子を複数持ちうるので，子を管理する場合は一次元配列より複雑な表現 (隣接リスト，隣接行列など 9.1) が必要となる．



練習問題 Marbles on a tree (Waterloo local 2004.06.12)

N 個の節点を持つ木が与えられる．木の各節点には，果物がない場合と，一つあるいは複数の果物がある場合がある．果物の合計は N 個である．各節点が一つづつ果物を持つようにするためには，最低何回の移動が必要か．一つの果物を一つの辺にそって移動すると 1 回と数える．

http://poj.org/problem?id=1909

入力のサンプルコードを示す:

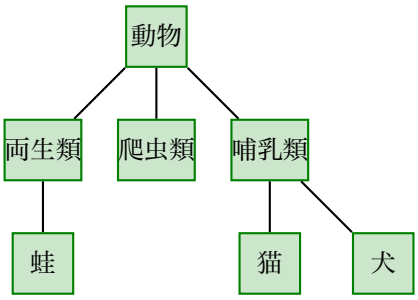
```
#include <cstdio>
using namespace std;
int N, P[10010], M[10010], V[10010];
int main() {
    while (~scanf("%d", &N) && N) {
        fill(P, P+N, -1);
        int /*葉の数*/L=0, id, c;
        for (int i=0; i<N; ++i) {
            scanf("%d", &id); --id; // [1,N] を [0,N-1] に
            scanf("%d %d", M+id, V+id); // marble の数, vertex の数
            for (int j=0; j<V[id]; ++j) {
                scanf("%d", &c); --c; // id(親) c(子供)
                P[c] = id;
            }
            if (V[id] == 0) ++L; // V が 0 なら id は葉
        }
        for (int i=0; i<N; ++i) {
            // 親を出力してみる
            printf("%d's parent is %d\n", i+1, P[i]+1);
        }
    }
}
```

(poj は cin を使うと時間制限になる問題があるため，scanf を勧める)

方針: 各節点について親から/へ移動する果物を考える．たとえば親から 3 つもらって 5 つ返すのは無駄で，それなら差し引き 2 つ送るだけで良い．つまり，各節点について，まずは子供の間で不足と余剰の調整を行ない，残った分を親に調整を依頼すると良い．

```
節点が調整済みかを表す配列を用意する．初めに葉を調整済みとしておく．
各節点での過不足を表す配列を用意する．初期値は，果物の配置数-1
while (根が調整済みでない) {
  for (全ての節点 i について) {
    if (i が調整済みでない かつ i の子供が全て調整済みなら) {
      i での過不足を親に押し付ける
      i を調整済みと記録
    }
  }
}
各節点での過不足の合計が答え
```

6.4.1 Lowest (Nearest) Common Ancestors



ある木について，木の二つの節点に共通する祖先でもっとも近い節点を lowest common ancestor と呼んで LCA と略す．図の例で，犬と猫の LCA は哺乳類．蛙と犬の LCA は動物．

練習問題 Nearest Common Ancestors (Taejon 2002)

一つの木と，その木の節点が二つ与えられる．二つの節点に共通するもっとも近い祖先を求めよ．

http://poj.org/problem?id=1330

いくつかのステップに分割して解いてみよう:

1. 入力を目で理解する． 一行目に数字 T があり,「木の情報と LCA を求める節点二つ」が T セット続いている．Sample Input の木を紙に書いてみる．

2. 木の入力を読み込む

```
#include <iostream>
using namespace std;
int N;
int P[10010];
int main() {
    int T;
    cin >> T;
    for (int t=0; t<T; ++t) { // t 番目の木について
        P[] を全て-1 に初期化
        cin >> N;
        for (int i=0; i<N-1; ++i) {
            int p, c;
            cin >> p >> c;
            P[c] の値を p に設定;
        }
         // この段階で
         //   P[根の番号] == -1
         //   P[c] == p  (c が親 p を持つとき)
        // という状態である
        int A, B;
        cin >> A >> B;  // LCA を求める二つの節点 A と B
        XXXXX
    } // 木に関するループ終了
}
```

“XXXXX” の部分で，各 P[] を表示して確認せよ．各節点の親は，紙に描いた木と一致するか?

3. LCA を求める準備として，ある節点 x から根までの節点を全て求めることを考える．考え方としては，x の親は P[x] で求められるから，親の親，親の親の親などとたどっていけばいずれは根に到達する．親をたどる際に節点を表示して，確認してみよう．

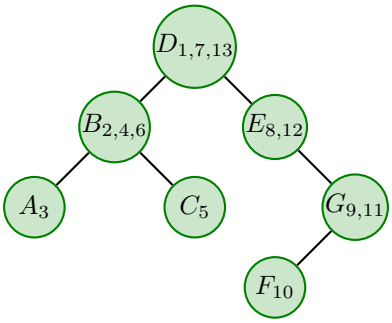
4. 節点 A と B のそれぞれについて，根までのパス (頂点番号列) の共通要素を求める．一番根から遠いものが求める答え．

5. POJ への提出．アカウントは上部の ページ “Register” から作成する．提出は，ページ下部の “Submit” から行う．

なお，同じ木に対して節点を変えながら何度も LCA を求める場合は，前処理をしておくことで一回あたり $O(\log N)$ で求めることができる．(参考書 p. 274)

6.5 木の話題

6.5.1 木のたどり方 (走査)



例 1: 数字は訪問順序を表す

根から始めて，下のような再帰的な手続きで，辺をたどりながら各頂点を一巡することを考える:

1. 左の子が存在し，かつ未訪問なら，左の子を訪問する
2. (そうではなくて) 右の子が存在し，かつ未訪問なら，右の子を訪問する
3. (そうではなくて) 親があれば，親へ戻る
4. いずれでもなければ，(根に帰ってきたので) 終了

図の「例 1」の木であれば，DBABCBDEGFGED と通る．例から分かるように，子供をもつ頂点は複数回 (正確には次数) 通過する．

この訪問順を基本としたうえで，各頂点を一度だけ処理する方法として，preorder (自分, 左, 右), inorder (左, 自分, 右), postorder (左, 右, 自分) などが用いられる．それぞれの方法では，子供と自分の優先順位が異なる．

	DBABCBDEGFGED
preorder	DBA C EGF
inorder	ABC DE FG
postorder	A CB FGED

練習問題 Tree Recovery (Ulm Local 1997)⋆

ある二分木について，preorder (root, left, right) で出力した頂点のリストと，inorder (left, root, right) で出力したリストが与えられる．(元の木を復元し) その頂点を postorder (left, right, root) で出力せよ．なお，元の木で，異なる頂点には異なるラベルがついている．

http://poj.org/problem?id=2255

ヒント:

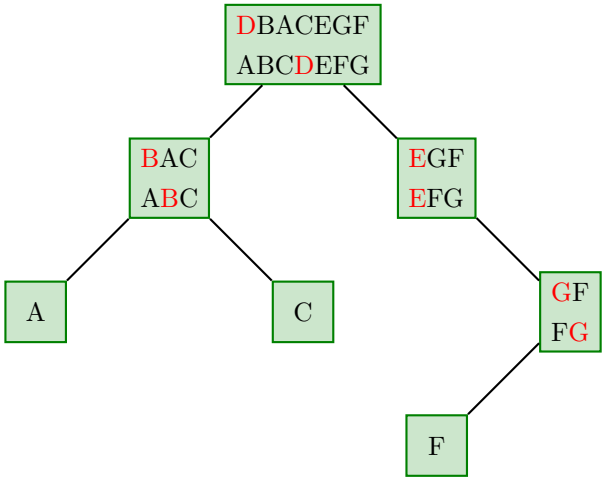
- ルートを A, その左側の部分木を L (null の場合もある), 右部分を R とする．preorder では，AL'R' のように並んでいる．(L' と R' はそれぞれの preorder 表記)．inorder では，L"AR" のように並ぶ (L" と R" はそれぞれの inorder 表記)
- preorder 表記の先頭から直ちに，ルート A が特定できる
- L, R 部分は，inorder 表記から A を探すことによって，要素を特定できる
- 文字数は，L' と L", R' と R" で等しいことに注意
- L' と L" から，左側の木を復元する問題は，元の問題の小さくなったバージョンである．

回答例:

```
#include <iostream>
#include <string>
using namespace std;
string preorder, inorder;
// preorder の [fp,lp) の範囲と，inorder の [fi,li) の範囲について
// 木を postorder で表示
void recover(int fp, int lp, int fi, int li) {
    int root;
    // preorder[fp] == inorder[root] となるような root を求める
    if (左側が存在すれば)
      recover(fp+1, fp+(root-fi)+1, fi, root); // 左側を表示
    if (右側が存在すれば)
      recover(fp+(root-fi)+1, lp, root+1, li); // 右側を表示
    cout << inorder[root]; // root を表示
}
int main() {
    while (cin >> preorder >> inorder) {
        recover(0, preorder.size(), 0, inorder.size());
        cout << endl;
    }
}
```

分割統治

上記の回答例は，扱う preorder と inorder の範囲を狭めながら，再帰的に処理を recover の処理を行う分割統治 (12 章) という考え方に基づいている．再帰関数/手続きについては，少し後 (6.5.1) に例を示す．呼び出し関係を図にすると，以下のようになる．図中，赤字が (その時点での部分木における) 根を表す．頂点内の文字列は，その頂点以下の部分木に対する，preorder 表記 (上段, fp と lp の範囲) と inorder 表記 (下段, fi と li の指す範囲)．



言語機能: 再帰, 文字列 (string)

再帰関数/手続き　階乗

```
int factorial(int n) { // 階乗
    if (n <= 1) return 1;
    return n*factorial(n-1); // 自分を呼び出す
}
```

使用例:

```
  cout << factorial(5) << endl;
```

Fibonacci 数

```
int fib(int n) {
    cerr << "computing fib(" << n << ")" << endl;
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-2)+fib(n-1);
}
```

使用例:

```
int main() {
  cout << "fib" << 30 << " = " << fib(30) << "\n";
}
```

文字列 (string)　C++の string クラスは，C の文字列よりもかなり便利なので，早めに慣れておくことをお勧めする．

```
#include <string>
  string word; // 定義
  string word2="ABCD"; // 定義と同時に初期化
  word = "EF"; // 代入
  string word3 = word + word2; // 連結
  char c = word[n]; // n 文字目を取り出す
  word[n] = 'K'; // n 文字目に代入 (n<word.size() でないと破綻)
  cin >> word; // 一単語読み込み (改行や空白文字で分割される)
  cout << word.size() << endl; // 文字数表示
  if (word.find("A") != string::npos) ... // もし word に A が含まれるなら...
```

6.5.2 木の正規化

練習問題 部陪博士，あるいは，われわれはいかにして左右非対称になったか (国内予選 2007)⋆⋆

式を表す二分木を与えられた規則で正規化せよ

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1152&lang=jp

(面倒なので経験者向け．なお，ICPC ではなるべく端末専有時間を短く解くことが求められる)

6.5.3 木の直径

練習問題 Fuel (Algorithmic Engagements 2011)⋆

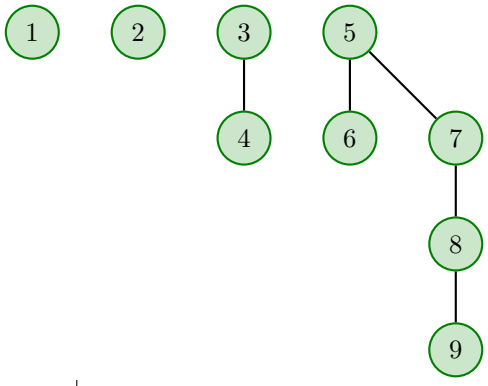
木と，歩いて良い辺の数が与えられるので，訪れる頂点の数を最大化した観光ツアーを設計せよ (walk; 同じ辺を複数回通って良い，始点と終点は別で良い)

http://main.edu.pl/en/archive/pa/2011/pal

第7章　Disjoint Setと全域木

7.1　Disjoint Set (Union-Find Tree)

要素同士が同じグループに属するかどうかを管理するデータ構造．グループ同士を併合する操作ができる．分離はできない (c.f. link-cut tree).



i	1	2	3	4	5	6	7	8	9
P[i]	1	2	3	3	5	5	5	7	8

$\{1\}, \{2\}, \{3, 4\}, \{5, 6, 7, 8, 9\}$ の 4 つのグループ

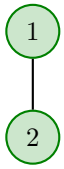
判定　同じ木に属するなら同じグループで，そうでなければ異なるグループである．それを，根 (root) を調べることで行う．例:

- **Q1. 6 と 8 は同じグループに属するか?**
  6 の属する木の根 (root) は 5 で，同様に 8 の根は 5 ➔ 同じグループ.
- **Q2. B と D は同じグループに属するか?**
  2 の属する木の根は 2 で，4 の根は 3 ➔ 異なるグループ.

併合　併合は，二つの節点を同一グループにまとめる操作で，一方の節点の属する木の根をもう一方の節点の属する木の根につなげることで実現する．どちらをどちらにつなげても，意味は変わらない．(小さな (低い) 木を大きな (高い) 木の下につける方が効率が良い，が，この資料では無視する)

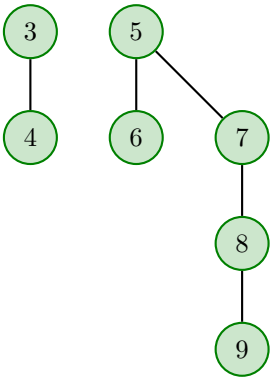
i	1	2
P[i]	1	2

元のグラフ



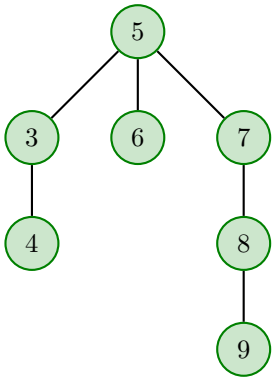
i	1	2
P[i]	1	1

1 と 2 を併合したグラフ



i	3	4	5	6	7	8	9
P[i]	3	3	5	5	5	7	8

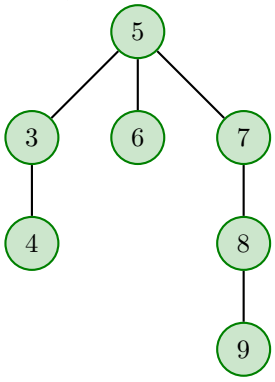
元のグラフ



i	3	4	5	6	7	8	9
P[i]	5	3	5	5	5	7	8

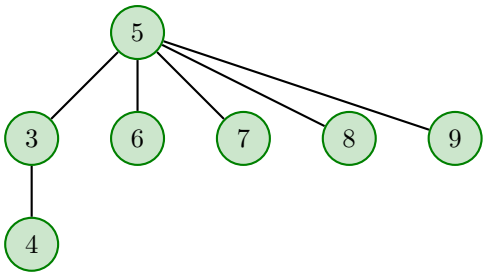
3 と 4 を併合したグラフ

効率化：パス圧縮　節点の根を求める操作は，根からの距離が遠いほど時間がかかる．そこで，なるべく根に直接つなげるような改善が有効である．



i	3	4	5	6	7	8	9
P[i]	5	3	5	5	5	7	8

元のグラフ



i	3	4	5	6	7	8	9
P[i]	5	3	5	5	5	5	5

root(9) = 5 を調べたついでに木を変形

コード例：　以下に，初期化，判定，併合のコード例を示す．なお，初見でコードの理解が難しい場合は，写経して動作を試した後に再度理解を試みるのが良い．(`rank` の概念は無視している．)

```
int P[10010]; // 0 から 10000 までの頂点を取り扱い可能
void init(int N) { // 初期化 はじめは全ての頂点はバラバラ
    for (int i=0; i<N; ++i) P[i] = i;
}
int root(int a) { // a の root(代表元) を求める
    if (P[a] == a) return a; // a は root
    return (P[a] = root(P[a])); // a の親の root を求め，a の親とする
}
bool is_same_set(int a, int b) { // a と b が同じグループに属するか?
    return root(a) == root(b);
}
void unite(int a, int b) { // a と b を同一グループにまとめる
    P[root(a)] = root(b);
}
```

実行例:

```
int main() {
  init(100);
  cout << is_same_set(1, 3) << endl;
  unite(1,2);
  cout << is_same_set(1, 3) << endl;
  unite(2,3);
  cout << is_same_set(1, 3) << endl;
}
```

上記のクラスカル法内で，辺 e を加える度に unite で辺の両端の頂点をグループ化する．辺 e の両端を a, b として，is_same_set(a,b) が真であれば辺 e を加えると閉路ができる (e と別に a, b パスがあるので)．

練習問題 Fibonacci Sets (会津大学プログラミングコンテスト 2003)

i 番目と j 番目 ($1 \leq i, j \leq V$) の Fibonacci 数である f[i] と f[j] 関して，f[i]%1001 と f[j]%1001 の差の絶対値が d 未満だったら，ノード i と j が同じグループに属するとする．V と d が与えられた時に，グループの数を数えよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1016

回答例:

```
int F[1001];
int main() {
    F[0] = 1, F[1] = 2;
    ... F[i] に i 番目の Fibonacci 数%1001 をあらかじめ計算, 代入しておく
    while (cin >> V >> D) {
        木を初期化
        for (int i=1; i<=V; ++i)
            for (int j=i+1; j<=V; ++j)
                if (F[j] と F[i] の差の絶対値が D 未満だったら)
                    グループ i とグループ j を同一化;
        i∈[1,V] に関して root(i) == i であるような数を数えて出力
    }
}
```

細かく作ってテストする

- Fibonacci 数の計算: F[2]..F[1000] までを for 文で代入する (4.4.1 の方針 3 で a[2] の代わりに F を用いることと，1001 の剰余を取りながら行う点が差)．つまり，i を小さい方から大きくしてゆけば，定

義通りに `F[i] = F[i-2]+F[i-1];` と計算できる．オーバーフローしないように計算途中でも 1001 の剰余を取る．(表示して確認する)

- Union-find 木を作る: 基本的には資料通り
  (初期化して，木を表示して確認する．いくつか併合しては木を表示して確認する．)
- Fibonacci 数での動作作成: 小さな V(たとえば 10) と適当な D を与えて，木を表示し，手計算と一致するかどうかをテストする．
- グループの数を数える: root(i) == i である個数ををテストする．

7.1.1 重み付き Union-find 木

練習問題 Never Wait for Weights (アジア地区予選東京 2012)

(意訳) 要素が属するグループだけでなく，要素間の距離を管理せよ．

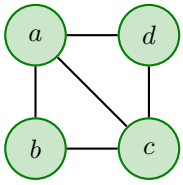
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1330

回答方針: 根との距離をデータに加えた Union-find 木を作れば良い．

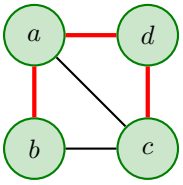
7.2 全域木

連結なグラフ G の頂点全てと辺の全てまたは一部分を用いて構成される木を**全域木 (spanning tree)** または**全点木**と呼ぶ．辺に重みがついている場合に，全域木に含まれる辺の重みの合計が最小であるような木を**最小 (重み) 全域木 (minimum spanning tree)** と呼ぶ．

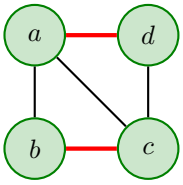
メモ: グラフが連結なら全域木が存在する．全域木は一つとは限らない (完全グラフの場合は n^{n-2} 個もある)．最小全域木も一つとは限らない．最小全域木を求める問題は，最小全域木のうちの一つを求める問題を指すことが多い．



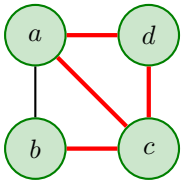
元のグラフ



赤い部分グラフは全域木



全域木でない (非連結)



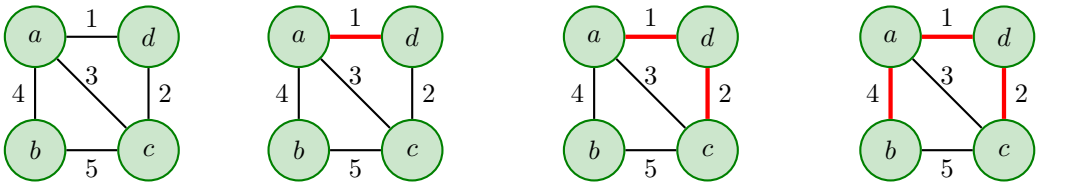
全域木でない (閉路)

7.2.1 クラスカル法

最小重み全域木を求める方法として，有名な方法にプリム法とクラスカル法があるが，ここでは後者を紹介する．

クラスカル法は以下のように動作する (参考書 p.101-):

- 辺を重みの小さい順にソートする
- T を作りかけの森 (最初は空，閉路を含まないグラフ，連結とは限らない) とする
- 重みの小さい順に，各辺 e に対して以下の操作を行う
  $T + e$ (グラフ T に辺 e を加えたグラフ) が閉路を含まなければ T に e を加える



元のグラフ　辺 ad (重み 1) を採用　辺 cd (重み 2) を採用　辺 ab (重み 4) を採用

$T + e$ が閉路を含むかどうかを効率的に判定する手法の一つが，union-find tree である．

練習問題 Stellar Performance of the Debunkey Family (PC 甲子園 2008)

問題: 街を連結に保ったまま，橋の維持費用を最小化したい．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0180

入出力例　入力を読み込んで，維持コストの少ない辺の順番に出力するコードは，たとえば以下のように作ることができる．今日の主眼は，クラスカル法の実習にあるので，これをそのままコピーしても問題ない．

```
#include <algorithm>
int N, M, A[10010], B[10010], COST[10010];
pair<int,int> bridge[10010]; // コストと橋番号のペア
int main() {
    while (N と M を読み込み，N が 0 より大きければ処理する) {
        for (int i=0; i<M; ++i) {
            A[i] と B[i] と COST[i] を読み込む;
            bridge[i].first = COST[i];
            bridge[i].second = i;
        }
        sort(bridge, bridge+M); // コストの小さい順に整列
        for (int i=0; i<M; ++i) {
            int cost = bridge[i].first;
            int a = A[bridge[i].second];
            int b = B[bridge[i].second];
            「a から b に cost の橋がかかっている」と表示
        }
    }
}
```

コード中の pair<int,int>とは struct pair { int first, second; }; に相当．

回答例:　上記の準備ができているとして，回答の骨子は以下のようになる．

```
        int 合計 = 0;
        for (int i=0; i<M; ++i) { //安い橋から順に
            if (端の両端の節点が既に連結だったら) continue;
            橋の両端の節点を同一グループに併合する;
            合計に，橋のコストを加える;
        }
        合計を出力;
```

7.2.2　複数の最小重み全域木

同じ重みの辺が存在する場合は，最小重み全域木は複数通り存在しうる．

練習問題 Byteland (1st Junior Polish Olympiad in Informatics)⋆

各辺毎に minimum spanning tree に含まれうるかを調べる

http://main.edu.pl/en/archive/oig/1/baj

時間制限は最大 15 秒確保されれているようだが，1.5 秒くらいで解ける．クラスカル法が正しい解を与える証明と関連.

第8章　動的計画法 (1)

概要

ある種の問題では，小さな問題を予め解いて表に覚えておくことで，効率的に解くことができる場合がある．動的計画法は，問題が持つ部分構造最適性を利用して効率の良い計算を実現する方法であるが，定義は後回しにして，その一部を見てみよう．

8.1　経路を数える

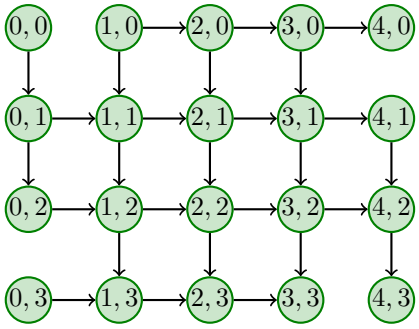
練習問題 平安京ウォーキング (UTPC2009)

グリッド上の街をスタートからゴールまで，(ゴールに近づく方向にのみ歩く条件で)
到達する経路を数える．ただし，マタタビの落ちている道は通れない．

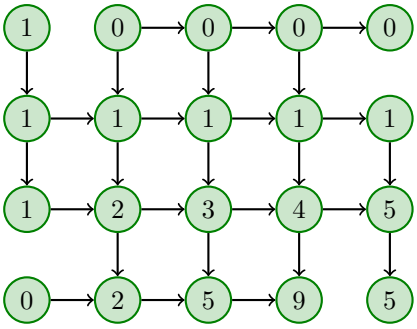
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2186

なお，マタタビがなければ組み合わせ (目的地までに歩く縦横の道路の合計から縦の道路をいつ使うか) の考え方から，直ちに計算可能である．

マタタビがある場合は，交差点毎に到達可能な経路の数を数えてゆくのが自然な解法で，サンプル入力3つめの状況は，下の図のようになる．



交差点の座標と通れる道 (右または下に移動可)



各交差点に到達可能な経路の数

ある交差点に (x, y) に到達可能な数 $T_{x,y}$ は，そこに一歩で到達できる交差点 (通常は左と上) 全ての値が定まっていればそれらの和として計算可能である．

$$T_{x,y} = \begin{cases} 0 & (x,y) \text{ が範囲外} \\ 1 & (x,y) = (0,0) \\ 0 & \text{上にも左にもマタタビ} \\ T_{x-1,y} & \text{上のみにマタタビ} \\ T_{x,y-1} & \text{左のみにマタタビ} \\ T_{x-1,y} + T_{x,y-1} & \text{上にも左にもマタタビなし} \end{cases}$$

マタタビの入力が多少冗長な形式で与えられるので，以下のように前処理して，移動不可能なことを示す配列などに格納しておくと使い勝手が良い．

- x 座標が同じ – (x1, max(y1,y2)) には上から移動不可
- y 座標が同じ – ((max(x1,x2), y1) には左から移動不可

8.2 最適経路を求めて復元する

練習問題 Spiderman (Tehran 2003 Preliminary)

指定の高さ H[i] だけ登るか降りるかを繰り返すトレーニングメニューを消化して地面に戻る．メニュー中で必要になる最大の高さを最小化する．

http://poj.org/problem?id=2397

この問題では，合計値ではなく最小値が必要とされる．

i 番目の上下移動を H_i(i は 0 から)，i 番目のビルで高さ h で居るために必要な最小コスト (=経路中の最大高さ) を $T_i[h]$ とする．それらの値を $T_0[0] = 0$(スタート時は地上に居るので), 他を ∞ で初期化した後，隣のビルとの関係から T_i と T_{i+1} を順次計算する．

$$T_{i+1}[h] = \min \begin{cases} \max(T_i[h - H_i], h) & \cdots i \text{ 番目のビルから登った場合 } (h \geq H_i) \\ T_i[h + H_i] & \cdots \text{同降りた場合} \end{cases}$$

ゴール地点を M として，ゴールでは地上に居るので，$T_M[0]$ が最小値を与える．

地面に，めりこむことはないので，非負の高さのみを考える．また登り過ぎるとゴール地点に降りられなくなるので適当な高さまで考えれば良い．

さらに，この問題では最小値だけではなく最小値を与える経路が要求される．どちらかの方法で求められる:

1. 最小値 $T_M[0]$ を求めた後，ゴールから順にスタートに戻る．隣のビルから登ったか降りたかは，T_i と T_{i+1} の関係から分かる．(両方同じ値ならどちらでも良い).
2. $T_{i+1}[h]$ を更新する際に，登ったか降りたかを $U_{i+1}[h]$ に記録しておく．$T_M[0]$ を求めた後に，$U_M[0]$ から順に $U_{M-1}[H_0]$ までたどる

8.3 区間を分割する

練習問題 輪番停電計画 (国内予選 2011)⋆

上手に区間を分割する．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1176&lang=jp

長方形に対する値を管理するタイプ．

練習問題 Ploughing (13th Polish Olympiad in Informatics)⋆⋆

畑を，縦 (幅 1 列) か横 (1 行) にスパっと切りとることを繰り返して，区分けする．どの範囲も数の合計が K 以下になるようにする．条件を満たす最小の分割数を求めよ．

http://main.edu.pl/en/archive/oi/13/ork

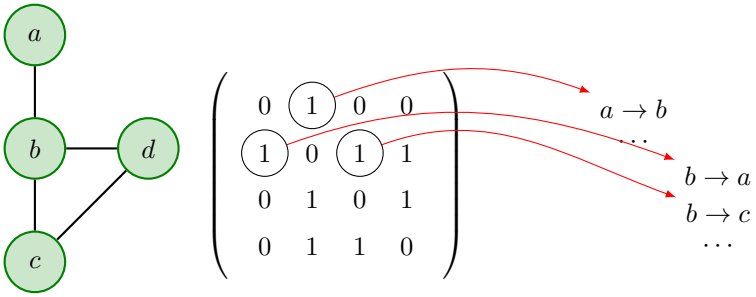
第9章　グラフの探索

概要

連結なグラフの辺を全てたどる方法として，幅優先探索 (BFS) と深さ優先探索 (DFS) を紹介する．

9.1　グラフの表現: 隣接行列

この資料では，グラフを隣接行列 (adjacency matrix) で表現する．節点 i から j への辺が存在する時，行列の i 行目 j 列目の値を 1 に，そうでないときに 0 とする．辺に向きのない，無向グラフを扱う場合には行列は対称になる．



左のグラフの a, b, c, d をそれぞれ $0, 1, 2, 3$ の数値に対応させると，隣接行列は右のようになる．すなわち $0, 1, 2, 3$ 行目が a, b, c, d から出る辺を表し，$0, 1, 2, 3$ 列目が a, b, c, d に入る辺を表す．

なお，辺のコストや経路の数などを表すために，行列の要素に 1 以外の値を今後使うこともある．

また，グラフの表現の中で隣接行列は以下の観点で比較的「贅沢な」表現方法である．都市の数を N とすると，常に N^2 に比例するメモリを使用する．グラフが疎な場合，すなわち辺の数が N^2 よりもかなり少ない場合は，隣接行列で値が 0 である要素が多くなり無駄が多い．たとえば「木」の場合は，辺の数は $N-1$ しかない．また，現実のグラフも電車の路線図のように疎であることが多い．そのような場合は，隣接リストなどの表現を検討すると良い．(参考書 p.90,91)

9.2　例題

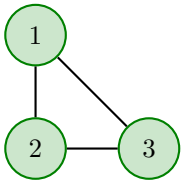
練習問題 A Bug's Life (TUD Programming Contest 2005)

性別の分からない虫がいる.「虫 i と虫 j の性は反対である」という情報が与えられた時に，矛盾しないかどうかを答えよ．

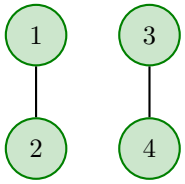
(または) グラフが二部グラフになっているかどうかを答えよ

http://poj.org/problem?id=2492

サンプル入力の解釈:



不整合



整合

データの保持

```
// 問題分で与えられる最大数
int bugs, edges;
// 隣接行列: e[i][j] が true なら, i<->j に辺がある
bool e[2010][2010];
// 各虫の色 0: 未定, 1,-1: 男女
int color[2010];
```

入出力例:

```
// 虫 id を id_color 色に割り当てて整合するかどうかを返す
// 整合する => true, しない => false
bool search(int id, int id_color) {
  ここを 3 種類作る
}
```

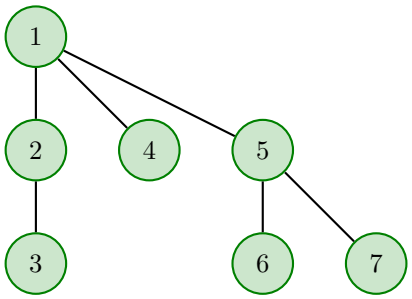
```
int main() {
    int scenarios;
    scanf("%d", &scenarios);
    for (int t=0; t<scenarios; ++t) {
        // この for 文のブロックが一つの問題
        fill(&e[0][0], &e[0][0]+2010*2010, 0); // 0 に初期化
        fill(&color, &color+2010, 0); // 0 に初期化
        scanf("%d %d", &bugs, &edges);
        for (int j=0; j<edges; ++j) {
            int src, dst;
            scanf("%d %d", &src, &dst);
            e[src][dst] = e[dst][src] = 1; // 両方向に通行化
        }
        bool ok = true;
        for (int j=1; j<=bugs; ++j) // 全ての虫について
            if (color[j] == 0 && !search(j, 1)) {
                ok = false; // 一回でも失敗したら, 不整合
                break;
            }
        if (t)
            printf("\n");
        printf("Scenario #%d:\n", t+1);
        if (! ok)
            printf("Suspicious bugs found!\n");
        else
            printf("No suspicious bugs found!\n");
    }
}
```

問題文中に注意が有る通り, poj の cin は, scanf の 10 倍以上遅いので, scanf を使う.

9.3　グラフの走査

Bug's life を解くには,「全ての辺を通」って，各頂点を 1 と-1 で塗り分けられる (隣接する頂点の数字が異なる) ことを確認すれば良い．グラフで,「全ての辺を通」る方法として，幅優先探索 (BFS) と深さ優先探索 (DFS) を紹介する．

Bug's life の sample 入力は小さすぎるため，少し複雑なグラフで動作を確認する．




```
1
7 6
1 2
1 4
1 5
2 3
5 6
5 7
```

9.3.1　幅優先探索 (BFS)

幅優先探索 (参考書 p. 36) は，出発地に近い頂点から順に訪問する．キュー (参考書 p. 32) というデータ構造を用いる．キューに入れた (push した) データは，front() 及び pop() の操作により早く入れた順に取り出される．

```
#include <queue>
#include <iostream>
bool bfs(int src, int src_color) {
    cerr << "bfs root = " << src << endl;
    queue<int> Q; // 整数を管理するキューの定義
    Q.push(src);
    color[src] = src_color; // 出発点に色を塗る
    while (! Q.empty()) {
        int id = Q.front(), id_color = color[id];
        Q.pop();
        // 動作確認用表示
        cerr << "visiting " << id << ' ' << id_color << endl;
        for (int j=1; j<=bugs; ++j) {
            if (! e[id][j]) continue;
            // id から到達可能な j 全てについてなにかする
            if (color[j] != 0) {
               // この場合は追加で処理が必要となるが一旦省略
               continue; // 訪問済みなら訪問しない (*)
            }
            color[j] = -id_color; // 新しい色を塗って
            Q.push(j); // todo 一覧に加える
        }
    }
    return true;
}
```

実行例:

```
bfs root = 1
visiting 1 1
visiting 2 -1
visiting 4 -1
visiting 5 -1
visiting 3 1
visiting 6 1
visiting 7 1
```

節点の訪問順は，1,2,4,5,3,6,7 と，根から距離 1,2,3,... と順にたどるものとなる．

問: (*) の continue; を消すと，どのような挙動になるか? 予想をたてた上で，実際に実行し，動作を確かめよ．

9.3.2 深さ優先探索, DFS (スタック)

深さ優先探索 (参考書 p. 33) は，現在訪問中の頂点に近い頂点から順に訪問する．スタック (参考書 p. 32) というデータ構造を用いる実装を次に示す．スタックに入れた (push した) データは，top() 及び pop() の操作により遅く入れた順に取り出される．一目でわかるように，ほとんど差がない (が動作は異なる)．

```
#include <stack>
#include <iostream>
bool dfs(int src, int src_color) {
    cerr << "dfs root = " << src << endl;
    stack<int> S;
    S.push(src);
    color[src] = src_color;
    while (! S.empty()) {
        int id = S.top(), id_color = color[id];
        S.pop();
        // 動作確認用表示
        cerr << "visiting " << id << ' ' << id_color << endl;
        for (int j=1; j<=bugs; ++j) {
            if (! e[id][j]) continue;
            // id から到達可能な j 全てについてなにかする
            if (color[j] != 0) {
               // この場合は特別な処理が必要となるが一旦省略
               continue; // 訪問済みなら訪問しない (*)
            }
            color[j] = -id_color; // 新しい色を塗って
            S.push(j); // todo 一覧に加える
        }
    }
    return true;
}
```

実行例:

```
dfs root = 1
visiting 1 1
visiting 5 -1
visiting 7 1
visiting 6 1
visiting 4 -1
visiting 2 -1
visiting 3 1
```

節点の訪問順は，1,5,7,6,4,2,3 と右の子に行けるだけ行き，行き止まったら上に戻り (あれば) 次の右の子に降りる順となる．兄弟のうちどの子供が優先されるかは，各頂点の番号のつけかたと，[1,bugs] の区間でひと通り試す際の小さい順か大きい順にも依存する．

問: (*) の continue; を消すと，どのような挙動になるか? 予想をたてた上で，実際に実行し，動作を確かめよ．

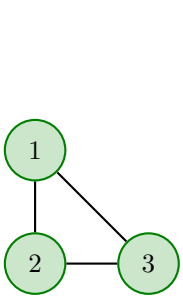
9.3.3 深さ優先探索, DFS (再帰)

DFS は再帰手続きを用いても実装できる．スタックデータ構造を明示的に用いる場合よりも，ソースコードはシンプルになりうる．

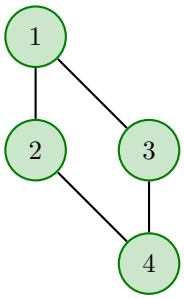
```
bool dfs(int id, int this_color) {
    if (color[id] != 0) // 既に色が塗られている
        return true; // (*) 実際には状況に応じて false を返す必要も
    color[id] = this_color;

    cerr << "visiting " << id << ' ' << this_color << endl;

    for (int j=1; j<=bugs; ++j) {
        if (! e[id][j]) continue;
        if (! dfs(j, -this_color))
            return false;
    }
    return true;
}
```

9.3.4 合流/ループの検査

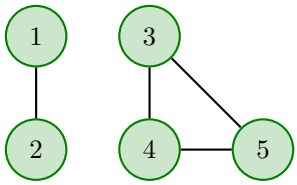
対象のグラフとして，これまでループのない木を例にとったが，一般のグラフでの動作を知るために，ループを持つ例を考える．例題では，奇数のループと偶数のループで動作を変える必要があるので二種類以上作る．




```
1
3 3
1 2
1 3
2 3
```




```
1
4 4
1 2
1 3
3 4
2 4
```




```
1
5 4
1 2
3 4
4 5
3 5
```

1. 前述の bfs や dfs に，上記のデータを与えて，どのような挙動 (頂点の訪問順) になるかを確認する
2. 上記のコード例内の，`if (color[j] != 0) {....}` の部分を適切に加筆することで，不整合の場合は false を返すようにせよ．

9.3.5 poj への提出

動作確認用の，cerr への出力は消しておくこと．

9.4 様々なグラフの探索

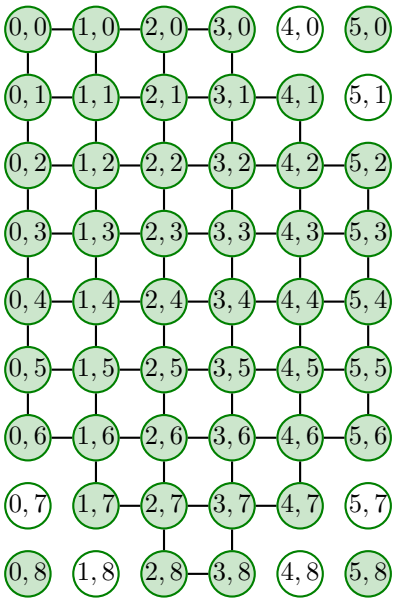
グラフの辺と頂点を問題文中で明示的に与えられない場合でも，自分でグラフを構成して幅優先探索 (BFS) あるいは深さ優先探索 (DFS) を行うことで解ける問題もある．

練習問題 Red and Black (国内予選 2004)

上下左右への移動だけで，行けるマスの数を求める．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1130&lang=jp

各マスを頂点として，移動可能な隣接する頂点同士に辺を張る．頂点に通し番号をつける必要はない．



マス (x,y) の上下左右　上下左右に隣接するマスは，(x+1,y), (x-1,y), (x,y+1), (x,y-1) の 4 つがありうる．但し，地図をはみ出していないかどうか注意が必要．

方向の表現　実際には，上下左右の移動を手で書くのはバグの元であるので避けたほうが良い．`const int dx[]={1,0,-1,0}, dy[]={0,-1,0,1};` のような配列を用意すると，探索内で似た様な 4 行が並ぶ部分を，for 文で纏めることができる．即ち，(x,y) の隣のマスの一つは，(x+dx[i], y+dy[i]) である．

マスに移動かどうか　(x,y) に行けるかどうか，(1) 地図をはみ出していなくて，(2) 壁でない，ことを調べる関数を作っておくと便利である．(1),(2) の順序でテストすること．

```
bool valid(int x, int y) {
   return x が [0,W] の範囲
        && y が [0,H] の範囲
        && (x,y) が壁でない;
}
```

回答骨子:　'@' の位置から，深さ優先探索あるいは幅優先探索で訪問できた頂点の数が答え．

練習問題 Curling 2.0 (国内予選 2006)⋆

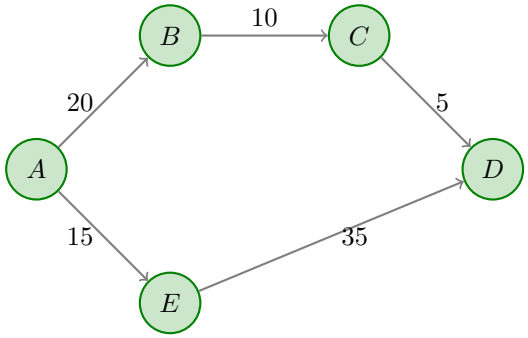
氷の上を滑らせながら最小何回でゴールに到達できるか，10 回以内にはできないかを求める．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1144&lang=jp

今回は，パックの位置が移動するだけでなく壁が壊れるので，両方をモデル化する必要がある．

第10章　最短路問題

こんな問題



A から D まで最も安くて幾らで行ける?
A..E は町．町をつなぐ道路は規定の料金がかかる．➔経路 {A,B,C,D} がコスト最小で 35

10.1　重み付きグラフと表現

辺に重みが付いたグラフ上の最短路問題を扱う．たとえば，グラフの節点が都市，辺が移動手段，辺についたコストが所要時間だとすれば，最短路問題は早く目的地に移動する問題と対応する．コストが通行料だとすると，最も安く目的地につく手段を求めることと対応する．辺に向きがある有向グラフの場合は，一方通行を表現可能である．なお，向きがない場合は，有向グラフで逆向きのコストが常に等しい特殊ケースと考えることができる．

なお，ここではコストは非負を仮定する．負のコストを用いて表現することが適切な例もある (たとえば，ある区間では通行料を払う代わりに，小遣いをもらえるスゴロクなど)．コストが負でありうる場合は非負の場合に成り立つ性質のいくつかは成り立たないため，注意が必要である．特に，コストの総和が負である閉路がある場合には，そこを回り続けるとコストは下がり続けるため，最短路は定義できない．

この章ではグラフの表現としては，もっとも簡単な隣接行列を用いる (9.1, 参考書 p.90,91)．隣接行列 K の i, j 要素 $K[i][j]$ は，i から j に有向辺があればそのコスト，ない場合は ∞ と表現する．

10.2　全点対間最短路

最短路問題を解くアルゴリズムには様々なものがあるが，まず全ての節点間の最短路を用いる Floyd-Warshall を覚えてくとよい (参考書 p.97)．見て分かるように `for` 文を 3 つ重ねただけの，簡潔で実装が容易なアルゴリズムである．

1: **procedure** FLOYD-WARSHALL(int K[][])　▷ i から j への最短路のコスト $K[i][j]$ を全て計算
　▷ 初期値は $K[i][j] = d_{ij}$ (i,j 間に辺がある場合)

```
                                          ▷ または K[i][j] = ∞ (ない場合)
2:    for k = 1..N do                     ▷ 都市番号が 1..N でない場合は，適宜変更すること
3:      for i = 1..N do                   ▷ 添字 k を i や j と入れ替えると動かないので注意!
4:        for j = 1..N do
5:          if K[i][j] > K[i][k] + K[k][j] then
6:            K[i][j] ← K[i][k] + K[k][j];      ▷ k を経由すると安い場合に更新
7:          end if
8:        end for
9:      end for
10:   end for
11: end procedure
```

動作の概略は以下の通り: 初期状態で $K[i][j]$ は直接接続されている辺のみ通る場合の移動コストを表す．アルゴリズム開始後，はじめに $k = 1$ のループが終了すると，$K[i][j]$ は，$i - j$ と移動する (直接接続されている辺を通る) か「$i - 1 - j$ と順に移動する」場合の最小値を表す．$k = 2$ のループが終了すると，$K[i][j]$ は，$i - j$ または $i - 1 - j$ または $i - 2 - j$ または $i - 1 - 2 - j$ または $i - 2 - 1 - j$ と移動するルートの最小値を表す．一般に $k = a$ のループが終了時点で $K[i][j]$ は，経由地として $1..a$ までを通過可能なパスのコストの最小値を表す．

証明概略

都市 a までを経由地に含む i から j の最短路 (の一つ) を D^a_{ij} と表記する．$a \geq 2$ の時 D^a_{ij} は，a を含む場合と含まない場合に分けられる．含まない場合は，都市 a に立ち寄っても遠回りになる場合で，D^{a-1}_{ij} と同一である．含む場合は，i から a を経由して j に到達する場合である．ここで，i から a までの移動や a から j までの移動で a を通ることはない (ものだけ考えて良い)．(各辺のコストが非負なので，最短路の候補としては，各都市を最大 1 度だけ経由するパスのみを考えれば十分である．) 従って i から a までの移動や a から j までの移動の最短路はそれぞれ，D^{a-1}_{ia} と D^{a-1}_{aj} である．a に立ち寄る場合と立ち寄らない場合を総合すると，$D^a_{ij} = \min(D^{a-1}_{ij}, D^{a-1}_{ia} + D^{a-1}_{aj})$ となる．

10.2.1 例題

練習問題 A Reward for a Carpenter (PC 甲子園 2005)

大工がどこかへ行って戻ってくる．(原文参照)

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0117

入出力　今回の入力はスペース区切りではなくカンマ (,) で区切られて与えられる．このようなデータを読む場合には `scanf` を用いると楽ができる．

C++で使う場合の注意点としては，`cstdio` を include することと，`scanf` を使う場合は `cin` は使わないこと．

```
#include <iostream>
#include <cstdio>
using namespace std;
int N, M, A, B, C, D, x1, x2, y1, y2;
int main() {
    scanf("%d%d", &N, &M);
    for (int i=0; i<M; ++i) {
        scanf("%d,%d,%d,%d", &A, &B, &C, &D);
        cerr << "read " << A << ' ' << B << ' ' << C << ' ' << D
             << endl;
        // A=>B がコスト C
        // B=>A がコスト D
    }
}
```

上限はいくつ? 街の数は最大 20 であるから，行列 K は十分に大きく設定する．注意点としては，街の番号は 1 から 20 で与えられることと，C++の配列の先頭は [0] であることである．今回は配列を大き目に確保して，必要な部分のみを使う ([0] は使わない) ことを勧める．

```
int K[32][32];
```

プログラムとして実装するうえでは ∞ として有限の数を用いる必要がある．この数は，(1) どのような最短路よりも大きな数である必要がある．最短路の最大値は全ての辺を通った場合で，各辺のコストの最大値と辺の数の積で見積もることができる． (2) 2 倍してもオーバーフローしないような，大きすぎない数である必要がある．(5,6 行目で加算を行うため)

```
const int inf = 1001001001;
```

多くの場合は 10 億程度の値を使っておけば十分である．(見積もりを越えないことを検算すること)

隣接行列の初期化 入力を読み込んで隣接行列を設定する部分をまず実装しよう．そして，隣接行列を表示する関数 `void show()` を作成し，表示してみよう．表示部分は前回の関数を流用可能である．ただし，今回は 0 列目と 0 行目は使わないことに注意.

サンプル入力に対しては以下の出力となることを確認せよ．(inf の代わりに具体的な数が書かれていても問題ない．また桁が揃っていなくても，自分が分かれば問題ない．)

```
inf   2   4   4 inf inf
  2 inf inf inf   3 inf
  3 inf inf   4 inf   1
  2 inf   2 inf inf   1
inf   2 inf inf inf   1
inf inf   2   1   2 inf
```

Floyd-Warshall 続いて，最短路のコストを計算する Floyd-Warshall を実装する．もっとも外側の `k` に関するループを行う度に，行列 K がどのように変化するかを確認すると良い．

最初のループ (`k=1`) 終了後

```
inf   2   4   4 inf inf
  2   4   6   6   3 inf
  3   5   7   4 inf   1
  2   4   2   6 inf   1
```

```
inf   2  inf  inf  inf   1
inf inf   2   1   2 inf
```

最終状態

```
  4   2   4   4   5   5
  2   4   6   5   3   4
  3   5   3   2   3   1
  2   4   2   2   3   1
  4   2   3   2   3   1
  3   4   2   1   2   2
```

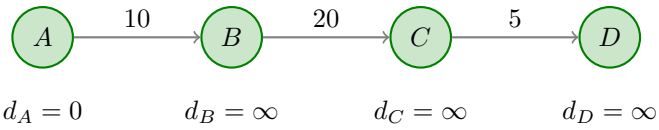
回答の作成　さて問題で要求されている回答は,「大工の褒美」であり，それは「柱の代金」-「殿様から大工が受け取ったお金」-「大工の町から山里までの最短コスト」-「山里から大工の町までの最短コスト」である．行列 K の参照と，適切な加減算で，回答を計算し出力せよ．

Accept されたら他の方法でも解いてみよう．

10.3　単一始点最短路

始点を一つ定めて始点から各点への最短路を求める問題は，全点対間で最小距離を求めるよりも効率的に解くことができる．往復のコストを求める場合は，単一始点最短路問題を 2 回解くと解が得られる．

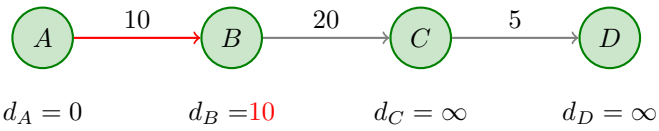
10.3.1　緩和 (relaxation)



はじめに，上のような単純なグラフを考え，A を始点として D までの距離を求める問題を考える．

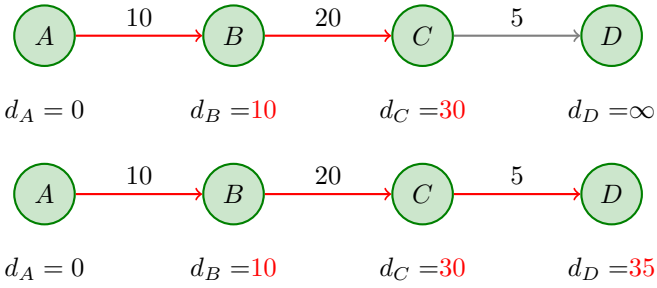
定義: $d[x]$ を A から x までの最短コストの上限とする．

初めに，$d[A] = 0$ (A から A まではコストがかからない)，$d[B] = d[C] = d[D] = \infty$ (情報がないので ∞) と定める．



定義:緩和操作

ある辺 (s,t) とそのコスト，$w(s,t)$ に対して，$d[t] > d[s] + w(s,t)$ である場合に，$d[t] = d[s] + w(s,t)$ と $d[t]$ を減らす操作を緩和と呼ぶ．$\delta[t]$ を t までの真の最小距離とすると $\delta[t] \leq \delta[s] + w(s,t)$ であるので，全ての節点で $\delta[n] \leq d[n]$ が保たれている状態で，この操作を行っても $\delta[t] \leq d[t]$ が保たれる．辺 AB に着目すると，$d[B] = min(d[B], d[A] + 10) = 10$ となり，$d[B]$ は ∞ から 10 に変化する．



同様に，$d[C] = min(d[C], d[B] + 20) = 30$，$d[D] = min(d[D], d[C] + 5) = 35$，と進めると D までの距離が求まる．$d$ が変化しなくなるまで緩和を繰り返すと，真の最短コストが得られる．どの順番に緩和を行うかで効率が異なる．以下，頂点の集合を V, 有向辺の集合を E, 辺 uv の重みを $w(u, v)$, 始点を v_s で表す．

10.3.2 Bellman-Ford 法

(参考書 p.95)

procedure BELLMAN-FORD($V, E, w(u, v), v_s$)
  for $v \in V$ **do**
    $d[v] \leftarrow \infty$ ▷ 初期化: 頂点までの距離の上限は ∞
  end for
  $d[v_s] \leftarrow 0$ ▷ 初期化: 始点までの距離は 0
  for $(|V| - 1)$ 回 **do** ▷ $|V|$ 回以上でも可
    for 辺 $(u, v) \in E$ **do**
      $d[v] \leftarrow \min(d[v], d[u] + w(u, v))$ ▷ 緩和
    end for
  end for
end procedure

- 緩和の順番は? – 適当に一通り
- いつまで続ける? 停まる? – (頂点の数-1) 回ずつ全ての辺について繰り返す
- 本当に最短? – 最短路の長さは最大 $|V| - 1$ であることから証明 (負閉路がない場合)

10.3.3 Dijkstra 法

(参考書 p.96)

procedure DIJKSTRA($V, E, w(u, v), v_s$)
  for $v \in V$ **do**
    $d[v] \leftarrow \infty$ ▷ 初期化: 始点から各頂点までの距離の上限は ∞
  end for
  $d[v_s] \leftarrow 0$ ▷ 初期化: 始点から始点までの距離は 0
  $S \leftarrow \emptyset$ ▷ 最短距離が確定した頂点の集合，最初は空
  $Q \leftarrow V$ ▷ 最短距離が未確定の頂点の集合，最初は全て
  while $Q \neq \emptyset$ **do** ▷ 最短距離が未確定の頂点がなくなるまで繰り返し

```
9:        select u s.t. arg min_{u∈Q} d[u]      ▷ 「最短距離が未確定の頂点」で d[u] が最小の u を選択
10:       S ← S ∪ {u}, Q ← Q \ {u}               ▷ u までの最小距離は確定
11:       for v ∈ Q s.t. (u, v) ∈ E do
12:           d[v] ← min(d[v], d[u] + w(u, v))   ▷ 緩和
13:       end for
14:   end while
15: end procedure
```

- 緩和の順番は? – 最短コストが確定している頂点 $u \in S$ から出ている辺の行き先で最もコストが低い頂点 v (線形探索または priority queue 等で管理)
- いつまで続ける? 停まる? – Q が空になると停止
- 本当に最短? – 背理法で証明 (w が非負の場合)

10.3.4 手法の比較

頂点の数を V とすると，Floyd-Warshall 法は，for 文の内側を見て分かる通り，V^3 回の基本演算が行われる．表 3.1 に当てはめると，$V = 100$ 程度であれば余裕であるが，$V = 1,000$ になるともう難しい．オーダ記法を用いると $O(V^3)$ となる．辺の数を E とすると，Bellman-Ford 法が $O(VE)$, Dijkstra 法が (実装によるが) $O(V^2)$ または $O(E \log V)$ 程度で，少し効率が良い．辺の数 E は，完全グラフでは V^2 程度，木に近い場合は V 程度なので，Bellman-Ford 法や Dijkstra 法が Floyd-Warshall 法よりどの程度早くなるかどうかはグラフの辺の数にも依存する．

10.4 練習問題

練習問題 崖登り (国内予選 2007)⋆

崖を登る最短の時間を求める

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1150&lang=jp

練習問題 Sums (10th Polish Olympiad in Informatics)⋆

整数の集合 A が与えられる．質問として与えられる数が，A の要素の和で表せるかどうかを答えよ．(正確な条件は原文参照)

http://main.edu.pl/en/archive/oi/10/sum

第11章　簡単な構文解析

こんな問題

- $(V|V)\&F\&(F|V)$ ➔F (真偽値の計算)
- 35=1?((2*(3*4))+(5+6)) ➔'+' (演算子の推定)
- 4*x+2=19 ➔x=4.25 (方程式を解く)
- C2H5OH+3O2+3(SiO2) == 2CO2+3H2O+3SiO2 (分子量の計算)

11.1　四則演算の作成

11.1.1　足し算を作ってみよう

大域変数:

```
const string S = "12+3";
size_t cur = 0; // 解析開始位置
int parse();
```

実行例:

```
int main() {
  int a = parse();
  assert(a == 15);
  assert(cur == S.size());
}
```

assertって何? (再掲)

ソースコード

```
#include <cassert>
int factorial(int n) {
  assert(n > 0); // (*)
  if (n == 1) return 1;
  return n * factorial(n-1);
}
```

実行例

```
cout << factorial(3) << endl; // 6 を表示
cout << factorial(-3) << endl; // (*) の行番号を表示して停止
```

足し算の (いい加減な) 文法

```
Expression := Number '+' Number
Number := Digit の繰り返し
Digit := '0' | '1' | ... | '9'
```

読みかた: (参考: (Extended) BNF)

- `P := Q` →P という名前の文法規則の定義
- `A B` →A の後に B が続く
- `'a'` →文字 a
- `x | y` →x または y

文法通りに実装する (Digit)

`Digit := '0' | '1' | ... | '9'`

```
#include <cctype>
int digit() {
  assert(isdigit(S[cur]));  // S[cur] が数字であることを確認
  int n = S[cur] - '0'; // '0' を 0 に変換
  cur = cur+1; // 一文字進める
  return n;
}
```

文法通りに実装する (Number)

`Number := Digit の繰り返し`

```
int number() {
  int n = digit();
  while (cur < S.size() && isdigit(S[cur])) // 次も数字か 1 文字先読
    n = n*10 + digit();
  return n;

}
```

文法通りに実装する (Expression)

`Expression := Number '+' Number`

```
int expression() {
  int a = number();
  char op = S[cur];
  cur += 1;
  int b = number();
  assert(op == '+');
  return a + b;
}
```

足し算だけならこれで動くはずである:

```
const string S = "12+3";
size_t cur = 0; // 解析開始位置
..
int parse() { return expression(); }
int main() {
  int a = parse();
  cout << a << endl; // 15 が出力されるはず;
}
```

テスト　“12+5” 以外にも “1023+888” など試してみよう

拡張: 引き算を加えよう

“12+5” を “12-5” としてみよう

方法: expression 関数で op が’+’ か’-’ を判定する

```
  if (op == '+') return a + b;
  else return a - b;
```

(assert も適切に書き換える)

拡張: 3 つ以上足す

“12+5” を “1+2+3+4” としてみよう

expression を書き換えて，複数回足せるようにする

```
int expression() {
    int sum = number();
    while (S[cur] == '+' || S[cur] == '-') { // 足し算か引き算が続く間
        char op = S[cur];
        cur += 1;
        int b = number();
        if (op == '+') sum に b をたす;
        else sum から b を引く;
    }
    return sum;
}
```

次の拡張

- 掛け算，割り算に対応:
  演算子の優先順位が変わるので新しい規則を作る
- (多重の) カッコに対応:
  同新しい規則を作って再帰する

11.1.2 カッコを使わない四則演算の (いい加減な) 文法

四則演算の (いい加減な) 文法

```
Expression := Term { ('+'|'-') Term }
Term := Number { ('*'|'/') Number }
Number := Digit { Digit }
```

読みかた: (参考: (Extended) BNF)

- A B ➔A の後に B が続く
- {C} ➔C の 0 回以上の繰り返し

例: 5*3-8/4-9

- Term: 5*3, 8/4, 9
- Number: 5, 3, 8, 4, 9

四則演算の実装 (Term)

```
Term := Number { ('*'|'/') Number }
```

```
int term() {
  int a = number();
  while (cur < S.size()
       && (S[cur] == '*' || S[cur] == '/')) {
    char op = S[cur++];
    int b = number();
    if (op == '*') a *= b; else a /= b;
  }
  return a;
}
```

四則演算の実装 (Expression)

```
Expression := Term { ('+'|'-') Term }
```

```
int expression() {
  int a = term();
  while (cur < S.size())
     && (S[cur] == '+' || S[cur] == '-')) {
    char op = S[cur++];
    int b = term();
    if (op == '+') a += b; else a -= b;
  }
  return a;
}
```

11.1.3 四則演算: カッコの導入

式全体を表す Expression が，カッコの中にもう一度登場 ➔再帰的に処理

カッコを導入した文法

```
Expression := Term { ('+'|'-') Term }
Term := Factor { ('*'|'/') Factor }
Factor := '(' Expression ')' | Number
```

factor() の実装例は以下:

```
int expression(); // 前方宣言
int factor() {
  if (S[cur] != '(') return number();
  cur += 1;
  int n = expression();
  assert(S[cur] == ')');
  cur += 1;
  return n;
}
```

term() の実装も，文法に合わせて調整すること．

11.1.4 まとめ

実装のまとめ:

- 文法規則に対応した関数を作る
- 帰り値の型は解析完了後に欲しいものとする
  - 四則演算 ➔整数
  - 多項式 ➔各次数の係数
  - 分子式 ➔分子量，各原子の個数…

文法の記述:

- 注意点: 演算子の優先順位や左結合や右結合
- 制限: 1 文字の先読みで適切な規則を決定できるように (LL(1))

補足　P := A { '+' A } の繰り返しを再帰で記述する?

- P := P '+' A | A
  ➔このまま実装すると P でずっと再帰
- 右結合に変換すると一応解析可能
  - P := A P'
  - P' := '+' A P' | ϵ

(ϵ は空文字列)

11.2 練習問題

練習問題 Smart Calculator (PC 甲子園 2005)

電卓を作る

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0109&lang=jp

回答例:

```
/*const*/ string S; // 値を変更するので const 属性を削除
...
int main() {
    int N;
    cin >> N;
    for (int i=0; i<N; ++i) {
        cur = 0;
        cin >> S;
        S.resize(S.size()-1); // 最後の=を無視
        cout << expression() << endl;
    }
}
```

練習問題 如何に汝を満足せしめむ？ いざ数え上げむ… (国内予選 2008)

論理式を満たす変数の割り当て方を求める.

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1155&lang=jp

練習問題 Equation Solver (Ulm Local 1997)

簡単な方程式を解く

http://poj.org/problem?id=2252

回答例: 右辺と左辺をそれぞれ解析し，一次の係数と定数項を両辺で比較.

練習問題 Questions (Algorithmic Engagements 2008)⋆⋆

P 人の王子と魔法使いのそれぞれの知識状態を上手にシミュレートして，質問になんと答えるかを当てる.

http://main.edu.pl/en/archive/pa/2008/pyt

補足

- Limitations: に，可能な変数の組み合わせは最大 600 とか，m計算途中の変数の値は絶対値 100 万を越えないなど，重要なことが書いてある
- サンプル入力と解説で “S 1 7 All sons know that there are less than 3 golden crowns.” とあるが，すぐ後に “M 1 7” があるのでこの説明で正しい．そうでなければ変数 7 の実際の値は，“S 1 7” からは読み取れない.
- 担当者の回答は 160 行くらい.

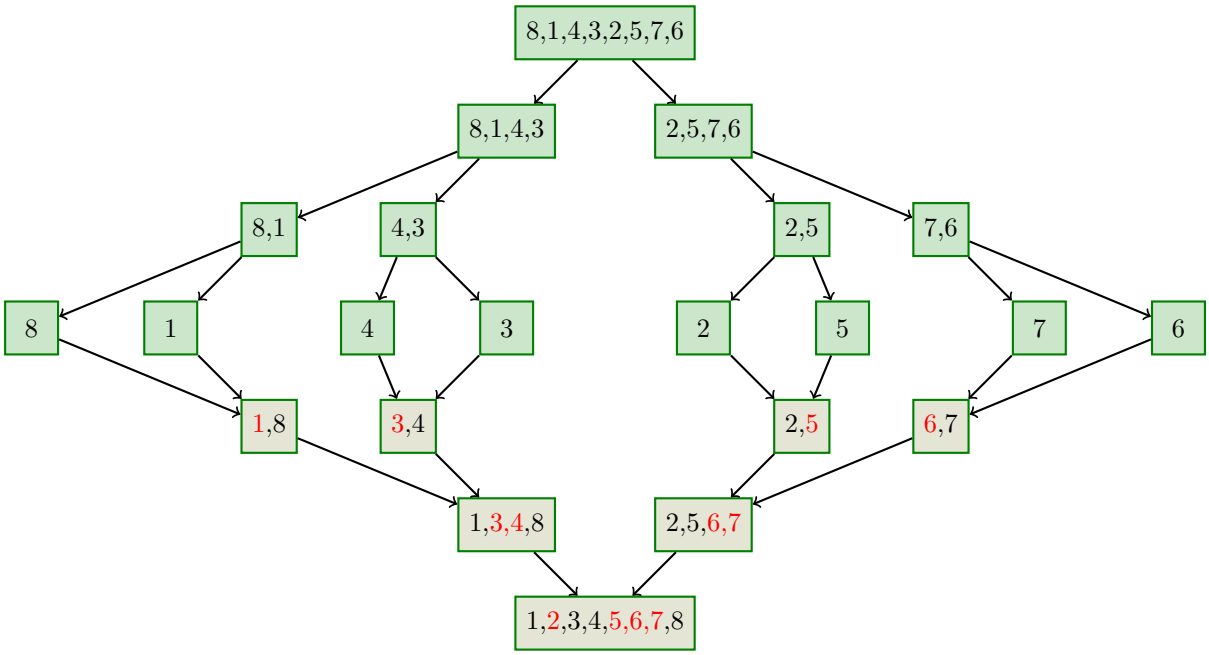
第12章　分割統治 (1)

概要

分割統治法は，問題を小さな問題に再帰的に分解してそれぞれを解いたうえで，順次それらを組み合わせて全体の解が得る技法を指す．小さな問題を扱う点は動的計画法と共通だが，分割統治では分割した問題に重なりがない場合を扱う．たとえば二分探索は分割統治の一つの手法であるが，フィボナッチ数の計算は分割統治とは通常呼ばれない．

12.1　Merge Sort

分割統治の例として，Merge Sort という整列手法がを取り上げる．この手法は，与えられた配列を半分に分割し，また元の大きさに組み立てる際に要素を整列する．図は，8,1,4,3,2,5,7,6 という配列を整列する際の処理の流れを示している．上半分で分割し，下半分で整列が行われる．赤字は併合の際に右側から来た要素を示す．具体的なソースコードは，次に紹介する `merge_and_count` とほぼ同様なので省略する．配列の要素数を N とすると，行が $O(\log N)$ あり，各行あたりで必要な計算が $O(N)$ なので，全体で $O(N \log N)$ という計算量が導かれる．



12.2 Inversion Count

配列内の要素のペアで，A[i] > A[j] (i<j) のものを数えたい．例えば配列 [3 1 2] の中には (3,1) と (3,2) の二つのペアの大小関係が逆転している．愚直に次のようなコードを書くと，要素数 N の自乗に比例する時間がかかる $O(N^2)$.

```
int N, A[128];
int solve() {
    int sum = 0;
    for (int i=0; i<N; i++) {
        for (int j=i+1; j<N; j++) {
            if (A[i] > A[j]) ++sum;
        }
    }
    return sum;
}
```

Merge sort の応用で，半分に分割しながら数えると，$O(N \log N)$ で求めることができる．

```
int N, A[たくさん]; // A は元の配列
int W[たくさん]; // W は作業用配列
int merge_and_count(int l, int r) { // range [l,r)
    if (l+1 >= r) return 0; // empty
    if (l+2 == r) { // [l,r) == [l,l+1] 要素 2 つだけ
        if (A[l] <= A[l+1]) return 0;  // 逆転はなし
        swap(A[l], A[l+1]);
        return 1; // 逆転一つ
    }
    int m = (l+r)/2; // [l,r) == [l,m) + [m,r)
    int cl = merge_and_count(l, m); // 左半分を再帰的に数える
    int cr = merge_and_count(m, r); // 右半分を再帰的に数える
    int c = 0; // 左と右を混ぜるときの逆点数
    int i=l, j=m; // i は [l,m) を動き，j は [m,r) を動いて，
    int k=l;      // 小さいものから W[k] に書き込む
    while (i<m && j<r) { // A[i] と A[j] を比べながら進む
        if (A[i] <= A[j]) W[k++] = A[i++]; // 左半分の方が小さく逆転なし
        else {
            W[k++] = A[j++];
            c += XXX; // 左半分の方が大きい，左半分で未処理の要素だけ飛び越える %m-i
        }
    }
    while (i<m) W[k++] = A[i++]; // 左半分が余った場合
    while (j<r) W[k++] = A[j++]; // 右半分が余った場合
    assert(k == r);
    copy(W+l, W+r, A+l);
    return cl + cr + c;
}
```

練習問題

練習問題 Bubble Sort (PC 甲子園 2007)

与えられた数列内の，大小関係が逆転しているペアの個数を求める

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0167

この問題は $O(N^2)$ でも通るので，愚直に解いた後，merge_and_count を試すのがおすすめ．

練習問題 Ultra-QuickSort (Waterloo local 2005.02.05)

類題: 数が大きいので long long を使うこと．こちらは $O(N \log N)$ が必要

http://poj.org/problem?id=2299

練習問題 Japan (Southeastern Europe 2006)⋆

長方形の島の東西に道路が走っている．交わる箇所の数を求めよ

http://poj.org/problem?id=3067

解き方の例: (東, 西) のペアでソートすると前 2 問と同じ問題になる．この問題は，累積和を管理しても解ける．

12.3 空間充填曲線

練習問題 Riding the Bus (世界大会 2003)⋆⋆

正方形内に描かれた Peano curve 上に格子点がある．与えられた二点間の距離を求めよ．距離とは，与えられた点から最短の格子点への距離 (複数ある場合は x,y 座標が小さいものまで) と，格子点間の Peano curve 上の道のり．(誤差の記述がちょっと心配)

https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=37&page=show_problem&problem=724

解き方の例: 全体を 9 分割して，関係ある場所だけを細かく探す．

第13章　おわりに

資料内の誤りの指摘や改善点などに気づかれた際は，筆者まで連絡いただけると幸いである．

筆者がコンテスト形式のプログラミングの存在を知ったのは，2004 年度の夏学期に，自主的な「勉強会」(冬学期以降は現「実践的プログラミング」) が増原准教授 (当時) により始められた時であった．それ以降，ACM-ICPC OB/OG 会の活動，ブログなどでの技術の交換，日本語のオンラインジャッジシステム (Aizu Online Judge) の公開，各大学でのオリジナルコンテストの開催，プログラミングコンテストチャレンジブックの出版など，日本の参加者のコミュニティの盛り上がりは記憶に新しい．一定以上のレベルの学習環境は現在までに充実していると思われるので，未経験者が楽しめるレベルに習熟するまでの効率を高めることができれば，より発展すると期待している．

付 録A　バグとデバッグ

プログラムを書いて一発で思い通り動けば申し分ないが，そうでない場合も多いだろう．バグを埋めるのは一瞬だが，取り除くには 2 時間以上かかることもしばしばある．さらに C や C++を用いる場合には，動作が保証されないコードをうっかり書いてしまった場合のエラーメッセージが不親切のため，原因追求に時間を要することもある．開発環境で利用可能な便利な道具に馴染んでおくと，原因追求の時間を減らせるかもしれない．特にプログラミングコンテストでは時間も計算機の利用も限られているので，チームで効率的な方法を見定めておくことが望ましい．

A.1　そもそもバグを入れない

逆説的だが，デバッグの時間を減らすためには，バグを入れないために時間をかけることが有効である．その一つは，良いとされているプログラミングスタイルを取り入れることである．様々な書籍があるので，自分にあったものを探すと良い．一部を紹介する．

- 変数を節約しない
  悪い例: (点 (x1,y1) と (x2,y2) が直径の両端であるような円の面積を求めている)

```
double area = sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))/2
 * sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))/2 * 3.1415;
```

  改善の例:

```
double radius = sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))/2;
double area = radius*radius*3.1415;
```

  なぜ良いか:

  - 人間に見やすい (x1 を x2 と間違えていないか確認する箇所が減る)
  - タイプ/コピーペーストのミスがない
  - 途中経過 (この場合は半径) を把握しやすい．(デバッガや printf で表示しやすい)

- 関数を使う

```
double square(double x) { return x*x; }
double norm(double x1, double y1, double x2, double y2) {
  return square(x2-x1)+ square(y2-y1);
}
double circle_area(double r) { return r*r*3.1415; }
```

- 定数に名前をつける

```
const double pi = 3.1415;
const double atan2(0.0,-1.0);
```

- 変数のスコープはなるべく短くする:
  変数のスコープは短ければ短いほど良い．関連して，変数の再利用は避け，一つの変数は一つの目的のみに使うことが良い．

```
int i, j, k;
// この辺に j や k を使う関数があったとする
...
int main() {
  for (i=0; j<5; ++i) // ああっ!
    cout << "Hello, world" << endl;
  ...
}
```

関数毎に必要な変数を宣言すると状況が大分改善する.

```
int main() {
  int i;
  for (i=0; j<5; ++i) // コンパイルエラー
    cout << "Hello, world" << endl;
}
```

しかし，C++ や Java など最近の言語では，for 文の中でループに用いる変数を宣言できるので，こちらを推奨する.

```
int main() {
  for (int i=0; i<5; ++i)
    cout << "Hello, world" << endl;
}
```

- ありがちな落とし穴をあらかじめ学び避ける
  C 言語 FAQ http://www.kouno.jp/home/c_faq/ 特に 16 奇妙な問題
- コンパイラのメッセージを理解しておく:
  - “if-parenth.cc:8:14: warning: suggest parentheses around assignment used as truth value”

    ```
    if (a = 1) return 1;
    if (a =! 1) cout << "ok";
    ```

  - “no return statement in function returning non-void [-Wreturn-type]”

    ```
    int add(int a, int b) {
      a+b; // 正しくは return a+b;
    }
    ```

A.2 それでも困ったことが起きたら

A.2.1 道具: assert

実行時のテストのために C, C++ では標準で assert マクロを利用可能である．assert 文は引数で与えられた条件式が，真であればなにもせず，偽の時にはエラーメッセージを表示して停止する機能を持つ.

コード例　階乗の計算:

```
int factorial(int n) {
  if (n == 1)
    return 1;
  return n * factorial(n-1);
}
int main() {
  cout << factorial(3) << endl; // 3*2*1 = 6 を出力
  cout << factorial(-3) << endl; // 手が滑ってマイナスをいれてしまったら，止まらない
}
```

上記の関数は，引数 n が正の時のみ正しく動く．実行時に，引数 n が正であることを保証したい．そのためには，cassert ヘッダを include した上で，assert 文を加える．見て分かるように assert の括弧内に，保証したい内容を条件式で記述する．

```
#include <cassert> // 追加
int factorial(int n) {
  assert(n > 0); // 追加
  if (n == 1)
    return 1;
  return n * factorial(n-1);
}
```

このようにして `factorial(-1)` 等を呼び出すと，エラーを表示して止まる．

```
Assertion failed: (n > 0), function factorial, file factorial.cc, line 3.
```

このように，何らかの「前提」にのっとってプログラムを書く場合は，そのことをソースコード中に「表明」しておくと見通しが良い．

`assert` は実行時のテストであるので，実行速度の低下を起こしうる．そのために，ソースコードを変更することなく assert を全て無効にする手法が容易されている．たとえば以下のように，cassert ヘッダを include する*前*に NDEBUG マクロを定義する

```
#ifndef NDEBUG
#  define NDEBUG
#endif
#include <cassert>
```

A.2.2 道具: _GLIBCXX_DEBUG (G++)

G++の場合，`_GLIBCXX_DEBUG` を先頭で define しておくと，多少はミスを見つけてくれる．(http://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html#debug_mode.using.mode)

```
#define _GLIBCXX_DEBUG
#include <vector>
using namespace std;
int main() {
    vector<int> a;
    a[0] = 3; // 長さ 0 の vector に代入する違反
}
```

実行例: (単に segmentation fault するのではなく，out-of-bounds であることを教えてくれる)

```
/usr/include/c++/4.x/debug/vector:xxx:error: attempt to subscript container
    with out-of-bounds index 0, but container only holds 0 elements.
```

A.2.3 道具: gdb

以下のように手が滑って止まらない for 文を書いてしまったとする．

```
int main() { // hello hello world と改行しながら繰り返すつもり
  for (int i=0; i<10; ++i) {
    for (int j=0; j<2; ++i)
      cout << "hello " << endl;
    cout << "world" << endl;
  }
}
```

gdb を用いる準備として，コンパイルオプションに`-g` を加える．

```
g++ -g -Wall filename.cc
```

実行する際には，gdb にデバッグ対象のプログラム名を与えて起動し，gdb 内部で run とタイプする

```
$ gdb ./a.out
(gdb が起動する)
(gdb) run (通常の実行)
(gdb) run < sample-input.txt (リダイレクションを使う場合)
...(プログラムが実行する)...
...(Ctrl-C をタイプするか，segmentation fault などで停止する)
(gdb) bt
(gdb) up // 何回か up して main に戻る
(gdb) up
#12 0x080486ed in main () at for.cc:6
6       cout << "hello " << endl;
(gdb) list
1 #include <iostream>
2 using namespace std;
3 int main() {
4   for (int i=0; i<10; ++i) {
5     for (int j=0; j<2; ++i)
6       cout << "hello " << endl;
7     cout << "world" << endl;
8   }
9 }
(gdb) p i
$1 = 18047
(gdb) p j
$2 = 0
```

主なコマンド:

- 関数の呼び出し関係の表示 bt
- 変数の値を表示: p 変数名
- 一つ上 (呼び出し元) に移動: u
- ソースコードの表示: list
- ステップ実行: n, s
- 再度実行: c
- gdb の終了: q

ソースコードの特定の場所に来た時に中断したり，変数の値が書き換わったら中断するようなこともできる．詳しくはマニュアル参照．

A.2.4 道具: valgrind

```
int main() {
    int p; // 初期化忘れ
    printf("%d\n", p);
}
```

gdb を用いる時と同様に-g オプションをつけてコンパイルする.

```
g++ -g -Wall filename.cc
```

実行時は，valgrind コマンドに実行プログラムを与える.

```
$ valgrind ./a.out
Conditional jump or move depends on uninitialised value(s)
...
```

A.3 標本採集: 不具合の原因を突き止めたら

バグの原因を特定したら，標本化しておくと将来のデバッグ時間を減らすための資産として活用できる.「動いたからラッキー」として先に進んでしまうと，何も残らない. 本筋とは離れるが，問題の制約を見落としたり，文章の意味を誤解したために詰まったなどの状況でも，誤読のパターンも採集しておくと役に立つだろう.

配列の境界

```
   int array[3];
   printf("%d", array[3]); // array[2] まで
```

初期化していない変数

```
   int array[3];
   int main() {
     int a;
     printf("%d", array[a]); // a が [0,2] でなければ不可解な挙動に
   }
```

return のない関数

```
int add(int a, int b) {
  a+b; // 正しくは return a+b;
}
int main() {
  int a=1,b=2;
  int c=add(a,b); // c の値は不定
}
```

stack 溢れ

```
  int main() {
    int a[100000000]; // global 変数に移した方が良い
  }
```

不正なポインタ

```
int *p;
*p = 1;

char a[100];
double *b = &a[1];
*b = 1.0;
```

文字列に必要な容量: 最後には終端記号'\0' が必要

```
   char a[3]="abc"; // 正しくは a[4] = "abc" もしくは a[] = "abc"
   printf("%s\n", a); // a[3] のままだと大変なことに
```

// A[i] (i の範囲は $[0, N-1]$) を逆順に表示しようとして

```
for (unsigned int i=N-1; i>0; ++i)
  cout << A[i] << endl;
```

// 整数を 2 つ読みたい

```
int a, b;
scanf("%d &d", &a, &b);
```