情報数理科学演習 II (学際科学科総合情報学コース)
広域システム特論 III(2) (学際科学科広域システムコース)

金子 知適 (kaneko@acm.org)，山口和紀

2018 年度 冬学期 (案) 再配布禁止
必ず授業当日に毎週ダウンロードし直すこと
Time-stamp: <2018-12-18 16:14:38 kaneko>

目次

第 1 章

ガイダンス・標準入出力を用いた自動テスト

2018-09-26 Wed

概要

情報数理科学演習 II では，アルゴリズムとデータ構造に関連した演習を行う．概念の理解だけでなく，正しく動作するプログラムを書ける能力を養うことを重視する．ただし，プログラムを書く能力はプログラミング演習 (夏学期) などで習得済みという前提で進める点．
一学期間で以下のトピックを扱う: 基本データ構造の活用 (stack, queue, string, priority quee, set, map), 動的計画法 (線形，二次元，木), グラフ (最小全域木，最短路，探索), 計算幾何，構文解析など．

1.1　オンラインジャッジで学ぶプログラミング

夏学期のプログラミング演習では，テストケースを自分で作成してプログラムの正しさを検証することを学んだ．冬学期は，インターネット上で稼働するオンラインジャッジを併用して，可能な範囲でテストの部分を自動化する．オンラインジャッジは，そこに提出されたプログラムを，予め作成されたテストケースを用いて判定 (ジャッジ) するサーバである．

1.1.1　先学期との違い

- 先学期は，関数の引数と計算結果の対応を直接テストした．
  例: `sum(2,3)==5`
  今学期は，様々なテストデータを試せるように，テストのためのデータを標準入力から読み込み，計算結果を出力するプログラムを作成する．

C++

```
1  int a,b;
2  cin >> a >> b; // たとえば 2 と 3 を読み込む
3  cout << sum(a,b) << endl; // 出力の正しさを入力作成者が確認
```

  変数 `a,b` は標準入力から読む．簡単には，キーボードの入力が標準入力に相当し，`2 3` と打ち込めば `sum(2,3)` をテストすることに相当する．なお，後述するリダイレクションという方法により，ファ

イルの内容を標準入力に与えることができる．自動テストにはこの方法が用いられる．

- 問題概要とともに，サンプル入出力 (少量のテストケース) が与えられる．
- ジャッジの入出力 (網羅的なテストケース) は，別にあり，通常は非公開で量も多い．
- プログラミング言語は，C++ を標準とする．C++ を思い出すこと．本日は C++ で行うこと．来週以降は特に断った場合を除いて，既に C++ に習熟している人は，Java や Ruby, Python などを使っても良い (eccs で動作しかつオンラインジャッジが受け付けるものであれば，何を使っても良い)．ただし，一部の問題は，実行速度の問題で C++ でしか，時間やメモリ制限内で処理が終わらない可能性がありうる．

1.1.2　アカウント作成と使い方

主に以下のオンラインジャッジを用いる．メインは AOJ である．

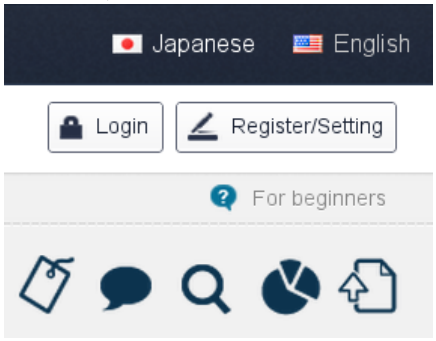
- Aizu Online Judge (AOJ) http://judge.u-aizu.ac.jp
- Peking University Judge Online for ACM/ICPC (POJ) http://poj.org
- Codeforces http://www.codeforces.com/
- szkopul.edu.pl (旧 MAIN.edu.pl http://main.edu.pl/en)

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.

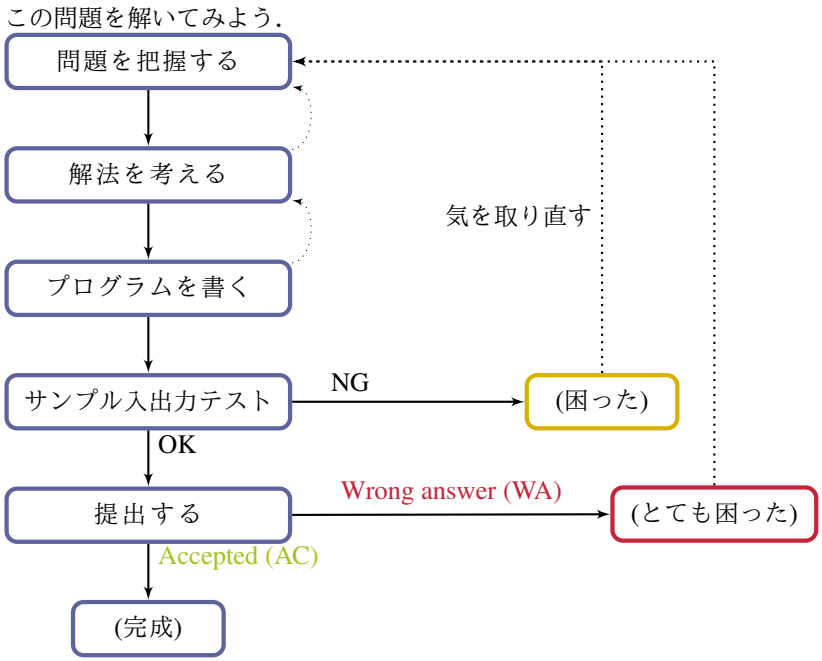
■正答でなかった時　様々な原因がありうるので，まずジャッジの応答がどれにあてはまるか，システムの使い方を誤解していないかなどを説明を読んで確認する．Submission notes (http://judge.u-aizu.ac.jp/onlinejudge/submission_note.jsp), Judge's replies (http://judge.u-aizu.ac.jp/onlinejudge/status_note.jsp), チュートリアル (http://judge.u-aizu.ac.jp/onlinejudge/AOJ_tutorial.pdf) などの資料がある．経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

1.2　実践例

例題　**X-Cubic**　(AOJ)

1 つの整数 x を読み，x の 3 乗を出力するプログラムを作成せよ

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_1_B



1.2.1　ファイルの作成

エディタ(Emacs, mi などお好みで) を起動し，以下のファイルを作り，xcubic.cc という名前で保存する．

C++

```cpp
#include <iostream>
using namespace std;
int main() {
  int x;
  cin >> x;
  cout << 2*2*2 << endl; // bug
}
```

1.2.2 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++ -std=c++11 -Wall xcubic.cc  # コンパイル
$ ./a.out  # 実行
```

#とそれより右は，コメントであり，入力する必要はない．-std=c++11 は，C++11 規格を有効にするオプションである．-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．特にプログラムの動作に疑問がある場合は，目立つ警告を解消してから質問すること．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．

1.2.3 実行・テスト

実行例: 以下，斜体はキーボードからの入力を示す．

```
$ ./a.out
2
8
```

($2^3 = 8$, 成功)

```
$ ./a.out
3
8
```

($3^3 = 27 \neq 8$, 失敗!)

1.2.4 ソースコード修正・再テスト

各自試すこと．
手元でうまく動くことを確認したら，AOJ に提出する．Accepted と判定されることを確認する．提出の際は，“C++11” または “C++14” を選択する．

1.3 標準入出力とリダイレクション

先の例題ではテストケースが数一つだったので，目で見れば事足りた．しかし，ある程度の複雑なプログラムをテストするには数一つでは不十分で，テストケースは大きくなる場合がある．たとえば，この問題は最大 1 万個の数値列を扱う．また出力も大きくなりうる．そのような際には，入力と，出力の検証を自動化すると良い．

(以下の資料では，意図的に誤りに誘導する部分があるが，気づいた場合もしばらくは声に出さないこと)

問題	**Min, Max and Sum**	(AOJ)

n 個の整数 $a_i (i = 1, 2, ...n)$ が与えられるので，それらの最小値，最大値，合計値を表示せよ

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_4_D

■入出力　以下のようなプログラムを作成する:

C++

```
1  /* minmaxsum.cc
2   */
3  #include <iostream>
4  using namespace std;
5  // 簡潔なスタイル
6  int N, A[10010]; // 少しだけ多めに取る
7  int min() {
8    return ...;
9  }
10 int max() {
11   return ...;
12 }
13 int sum() {
14   return ...;
15 }
16 int main() {
17   cin >> N;
18   for (int i=0; i<N; ++i)
19     cin >> A[i];
20   cout << min() << ' ' << max() << ' ' << sum() << endl;
21 }
```

変数 N や A は名前も含めて問題文に対応するものをなるべく使用し，さらに，本質的な部分に集中するために*global 変数*とすることを勧める．配列 A の長さは入力により異なるが，最大値はたかだか 1 万なので，ケチケチせずに最大値を確保する (気にする場合は vector を用いてもよいが，new や delete の使用は勧めない).

なお，実用的なプログラムを書く際は，変数のスコープを必要な範囲に狭めることが望ましい．必要であれば，以下のような書き換えられることを前提に，本演習では global 変数を用いる書き方を勧めている．

C++

```cpp
#include <vector>
#include <memory>
class MinMaxSumSolver { // 行儀の良いスタイル
  int N;
  std::vector<int> A;
public:
  void setup();
  int min() {
    return ...;
  }
  int max() {
    return ...;
  }
  int sum() {
    return ...;
  }
  void solve();
};
int main() {
  std::shared_ptr<MinMaxSumSolver> solver(new MinMaxSumSolver());
  solver->solve();
}
```

■リダイレクションを用いたテスト　まず sample-input.txt を，作成しておく．数字を 1 文字づつ入力するのではなく，コピーペーストを用いること (打ち間違いを防ぐため)．本資料では作成できていることを示すために cat の実行例を載せることがある．

```
$ cat sample-input.txt
5
10 1 5 4 17
```

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 < sample-input.txt
1 17 37
```

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 < sample-input.txt > my-output.txt
$ cat my-output.txt
1 17 37
```

2 つのファイル比較 (diff):

```
$ diff -u sample-output.txt my-output.txt
```

(差分がある場合のみ，出力がある)

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/

■ヒント　普通に組むと，Accepted ではなく Wrong Answer となる場合がある．

Run#	Author	Problem	Status	%	Lang	Time	Memory	Code	Submission Date
1515235	kaneko	ITP1_4_D: Min, Max and Sum	: Wrong Answer	18/20	C++	00:00 s	1200 KB	362 B	2015-09-16 10:50

中央付近の 18/20 という数字は，テストケースの 18 番目まで正答し，19 個目で失敗したという意味である．その部分がハイパーリンクになっているのでクリックすると，詳細が分かる．

Case #16:	: Accepted	00.00 sec	1168 KB
Case #17:	: Accepted	00.00 sec	1200 KB
Case #18:	: Accepted	00.00 sec	1196 KB
Case #19:	: Wrong Answer	00.00 sec	1200 KB

さらに Case #19:の行をクリックすると，実際のデータを見ることができる (問題による).

< prev | 19 / 20 | next > 04_maximum_02.in : Wrong Answer 00.00 sec 1200 KB

Judge Input #19 (in19.txt | 68926 B)

```
10000
430143 602887 783032 225925 905915 978433 239648 49
```

Judge Output #19 (out19.txt | 21 B)

```
28 999997 5019101515
```

このデータを手元で試してみよう．データは，コピーペーストするには大きすぎるので，まず in19.txt の部分をクリックしてダウンロードする．環境やブラウザによるが ITP1_4_D_in19.txt という名前でダウンロードフォルダに保存されたとすると，適宜，リダイレクションによって実行する．

```
$ ./a.out < ~/Downloads/ITP1_4_D_in19.txt
28 999997 724134219
```

また，今回はプログラムの出力と Judge Output との間に差があることは一目でわかるが，一般に 5019101515 のような桁数の数字を目で比較するのは困難であるから (1 文字異なっていても気づかない)，同様にダウンロードして diff コマンドにより比較するのが良い．

(原因と対策の解説は，来週にも行う予定だが，白文字でこちらにも記しておく．コピーペーストなどで読めるはずだが，読む前に，十分に=15 分間は仮説を検討すること．却下した仮説も含めて文字で記しておくと良い．)

1.4 本日の課題

- 問題を解いて，AOJ で Accepted を得たことを確認して，感想 (学んだことなど，特になければ合計所要時間と，どこにどの程度時間がかかったか) とともに，ITC-LMS に提出する．本日の到達目標は AOJ の使い方に一通り馴染むことであるので，発展課題は特に設けていない．

提出の注意:

- ソースコードにコメントを書き入れたテキストファイルを提出する (PDF, Microsoft Word, rtf などは避けること)．コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である．

C++

```
1  // minmaxsum.cc
2  #if 0
3    所要時間は..であった. ...を復習した.
4    ...
5  #endif
6  int main() {
7  }
```

- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．

1.A 指定したフォルダへのファイルの保存とコンパイル

冬学期もフォルダを活用して，ソースコードを整理することをすすめる．

各回では，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 Math-IS-II                                                          1
# (フォルダ作成，最初の一回のみ)                                             2
$ mkdir Math-IS-II/week1                                                    3
# (各週毎に行う)                                                           4
$ cd Math-IS-II/week1                                                       5
# (カレントディレクトリを変更．$HOME/Math-IS-II/week1/ 以下にソースコードを保存)  6
$ g++ -std=c++11 -Wall sample1.cc                                           7
$ ./a.out                                                                   8
# (実行)                                                                   9
```

1.B　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．

1.B.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)

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

  改善の例:

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

C++
```
1 double square(double x) { return x*x; }
2 double norm(double x1, double y1, double x2, double y2) {
3   return square(x2-x1)+ square(y2-y1);
4 }
5 double circle_area(double r) { return r*r*3.1415; }
```

- 定数に名前をつける

C++
```
1 const double pi = 3.1415;
2 const double pi = 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．

C++
```
1 int i, j, k;
2 // この辺に j や k を使う関数があったとする
3 ...
4 int main() {
5   for (i=0; j<5; ++i) // ああっ！
6     cout << "Hello, world" << endl;
7   ...
8 }
```

  関数毎に必要な変数を宣言すると状況が大分改善する．

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

  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．

C++
```
1 int main() {
2   for (int i=0; i<5; ++i)
3     cout << "Hello, world" << endl;
4 }
```

- 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`

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

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

C++
```
1 int add(int a, int b) {
2   a+b; // 正しくは return a+b;
3 }
```

1.B.2　それでも困ったことが起きたら

道具: 表示

プログラムの動作を把握するための，もっとも原始的な方法は，表示である．例として以下のプログラムを考える．

■コード例　階乗の計算:

C++
```
1  int factorial(int n) {
2    if (n == 1)
3      return 1;
4    return n * factorial(n-1);
5  }
6  int main() {
7    cout << factorial(3) << endl; // 3*2*1 = 6 を出力
8    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
9  }
```

上記の関数は，引数 n が正の時のみ正しく動く．誤って負の n で実行した場合には，実行が終わらない，segmentation fault で停止する，などで以上があることが観察される．しかし，原因が負の n にあることは，観察だけからは分からない．もし，原因の候補に心当たりがあるのなら，動作のキーとなる変数を表示することで，追加の情報を得ることが出来る．

C++
```
1  int factorial(int n) {
2    cerr << n << "\n"; // 追加
3    if (n == 1)
4      return 1;
5    return n * factorial(n-1);
6  }
```

このようにして factorial(-1) 等を呼び出すと，膨大な出力が得られることから，異常を検知する手がかりになる．しかし，このような表示と人の目による検証は，最善の方法とはいえない．

道具: assert

自動化された (人の目に頼らない) 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．

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．

C++
```
1  #include <cassert> // 追加
2  int factorial(int n) {
3    assert(n > 0); // 追加
4    if (n == 1)
```

```
5       return 1;
6     return n * factorial(n-1);
7   }
```

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.   1
```

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用意されている．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

C++

```
1  #ifndef NDEBUG
2  #  define NDEBUG
3  #endif
4  #include <cassert>
```

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)

C++

```
1  #define _GLIBCXX_DEBUG
2  #include <vector>
3  using namespace std;
4  int main() {
5      vector<int> a;
6      a[0] = 3; // 長さ 0 の vector に代入する違反
7  }
```

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 container1
    with out-of-bounds index 0, but container only holds 0 elements.       2
```

道具: gdb

利用環境

現在の教育用計算機システム (iMac 端末) では gdb を利用できない．各自 Ubuntu など Linux 環境を準備すれば標準で動作する．

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．

C++

```
1  int main() { // hello hello world と改行しながら繰り返すつもり
2    for (int i=0; i<10; ++i) {
3      for (int j=0; j<2; ++i)
4        cout << "hello␣" << endl;
5      cout << "world" << endl;
6    }
7  }
```

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．

道具: valgrind

利用環境

現在の教育用計算機システム (iMac 端末) では valgrind を利用できない．ssh サーバでは可能．https://www.ecc.u-tokyo.ac.jp/system/outside.html#ssh 各自 Ubuntu など Linux 環境を準備すれば標準で動作する．

C++

```
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)
...
```

現状の eccs の imac 端末では，valgrind が動作しないが，ssh サーバにログインすれば使用可能である．
http://www.ecc.u-tokyo.ac.jp/system/outside.html#ssh

1.B.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

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

初期化していない変数

C++
```
1     int array[3];
2     int main() {
3       int a;
4       printf("%d", array[a]); // a が [0,2] でなければ不可解な挙動に
5     }
```

return のない関数

C++
```
1  int add(int a, int b) {
2    a+b; // 正しくは return a+b;
3  }
4  int main() {
5    int a=1,b=2;
6    int c=add(a,b); // c の値は不定
7  }
```

stack 領域溢れ

C++
```
1    int main() {
2      int a[100000000]; // global 変数に移した方が良い
3    }
```

不正なポインタ

C++
```
1  int *p;
2  *p = 1;
3
4  char a[100];
5  double *b = &a[1];
6  *b = 1.0;
```

文字列に必要な容量: 最後には終端記号'\0' が必要

C++

```
1     char a[3]="abc"; // 正しくは a[4] = "abc" もしくは a[] = "abc"
2     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

C++

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

// 整数を 2 つ読みたい

C++

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

第 2 章

全探索と計算量の見積もり

2018-10-03 Wed

概要

可能性を全て試すプログラムと，その計算量の見積もりについて学ぶ．今回から一部の問題には考察課題が指定されている場合がある．ソースコードを提出する際に，指定された考察も添えること．

2.1 問題の大きさの把握 (Constraints)

例題 **Min, Max and Sum (再掲)** (AOJ)

n 個の整数 $a_i (i = 1, 2, ...n)$ が与えられるので，それらの最小値，最大値，合計値を表示せよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_4_D

入力の制約: $0 < n \leq 10\,000,\ -1\,000\,000 \leq a_i \leq 1\,000\,000$

実行時の制約: Time Limit : 1 sec, Memory Limit : 65536 KB

さて，求められているプログラムは「制約を満たす範囲でどのような入力にも正しい出力を行う」ものである．従って，扱う入力の範囲を理解し，適切な解法を設計する必要がある．問題の範囲と有効な解法の関係は，先学期のプログラミング演習では意図的に扱わなかったが，今学期の本演習では主題に近い重要な点である．

■数値の表現 通常，C++ で整数は `int` 型の変数で表現する．一方，この演習の iMac 環境では，表 2.1 のように，`int` 型や他の整数型で表現できる値の範囲には限りがある．

配列の各要素は，$\pm 1\,000\,000\ (\approx 2^{20})$ なので，int で表現可能な範囲に余裕を持って収まっている．一方，「合計」はどうだろうか．個数 n の上限ですべて要素が最大値をとったとすると

$$1\,000\,000 \cdot 10\,000 = 10^{10} > 2^{31} \approx 2 \cdot 10^9$$

と 32bit 整数で表現できる範囲を超える．つまり，合計を表す変数には long long 型を使う必要がある．このことをプログラムを書き始める前に把握できるようになることが，目標の一つである．[*1]

[*1] このような計算をスラスラと行うために $2^{10} = 1\,024 \approx 10^3$ を暗記しておくことを勧める．

表 2.1　この資料が想定する環境での，符号付き整数の表現

type	bits	下限	上限	備考
int	32	-2^{31}	$2^{31}-1$	約 20 億
long	32/64			使用を勧めない
long long	64	-2^{63}	$2^{63}-1$	C++11 から標準型，以前は gcc 拡張
__int128_t	128			gcc 拡張，portable でないが強力

各型で表現可能な範囲は環境によって異なりうる．使用中の環境については，以下のようなプログラムを用いて確認できる．numeric_limits の文法は割愛するが，興味のあるものはテンプレートクラスと標準ライブラリについて調べると良い [6].

C++

```
#include <limits>
#include <iostream>
using namespace std;
int main() {
  cout << sizeof(int) // バイト数
       <<'␣'<< numeric_limits<int>::digits // 符号以外の bit 数
       <<'␣'<< numeric_limits<int>::digits10 // 十進桁数
       <<'␣'<< numeric_limits<int>::min()
       <<'␣'<< numeric_limits<int>::max()
       << endl;
  cout << sizeof(long long)
       <<'␣'<< numeric_limits<long long>::digits
       <<'␣'<< numeric_limits<long long>::digits10
       <<'␣'<< numeric_limits<long long>::min()
       <<'␣'<< numeric_limits<long long>::max()
       << endl;
}
```

```
$ ./a.out
4 31 9 -2147483648 2147483647
8 63 18 -9223372036854775808 9223372036854775807
```

なお表 2.1 には符号付き整数のみ記載したが，符号なし整数 (unsigned int など) についても，各自把握のこと．C++11 では#include<cstdint>すると，int32_t, int64_t などの型を使用可能である．将来はこちらを使用するべきだが，本資料では，int と long long を用いる．

■時間と空間の制約　各問題には，実行時の制約も付加されている．先の例題ではプログラムは，実行時間 1 秒未満で回答を終えること，利用メモリは 64MB 未満であることが求められている．

プログラムの実行時間やメモリの使用量は，(1) 実行環境，(2) 問題の大きさ，(3) アルゴリズムによって大きく変わりうる．

それぞれ簡単に議論すると，AOJ の実行環境 [1] はサイト内の文書に記載されているが，演習室の端末よりも高速なので通常は気にする必要はない．つまり手元で十分効率的に動くことを確認すれば良い．

問題の大きさ [2] は，たとえばこの例題では数値列の長さが相当する．数値をすべて標準入力から読み配列に格納するとすると，長さに比例する時間がかかると予想される．たとえば，数値列が長さ 1 であればすぐに終わるが，10 000 であれば (正確に 1 万倍かどうかはさておき) もっと時間がかかるだろう．使用メモリは，数値を整数型 (4 byte) で表すと $4 \cdot 10\,000$ byte であり，入力をすべて配列に格納しても 1MB に達しない，微々たる量であることが分かる．(プログラムが使用するメモリ量は他にもあるが，この演習では問題の大きさで変化しうる部分のみを考える．)

アルゴリズム [3] がどのような役割を果たすかが，もっとも重要な話題であり，今後時間をかけて取り扱う．

2.2 組み合わせを試す (generate and test)

先週の問題で要素を全て調べて最大値を求めたように，コンピュータが得意な解法は全ての可能性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．全ての可能性を列挙するには，for 文を用いたり，再帰を用いたりする．

問題 **Tax Rate Changed** (国内予選 2014)

2 つの商品の消費税率変更前の税込合計価格を元に， 新消費税率での税込合計価格が最大いくらになるかを計算するプログラムを作って欲しい． 1 円未満切り捨て等の細かい条件は問題文を参照のこと． http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1192&lang=jp

制約 (抜粋): $10 < s < 1000$， 商品の税抜価格は 1 円から $s - 1$ 円のすべてを考慮に入れる

Time Limit : 8 sec, Memory Limit : 65536 KB

(1) 2 つの商品の価格の候補の組み合わせを全通り試す．(2) 候補が，変更前の税率で合計価格になっているかを調べる．そうでなければ捨てる (3) 変更後の税率での合計価格を計算する．最大値なら記憶する．という方針で作ってみよう．

仮に税込みの価格を計算する tax(*rate, price-without-tax*) という関数があったとすると，次のような構造になるだろう．問題文では小文字の s を使っているが，本資料のプログラムでは (i, j, k などの変数と区別して問題文由来である目印に) 大文字で表記する．

C++

```
1  int maximum = 0;
2  for (int i=1; i<S; ++i)
3    for (int j=i; j<S; ++j)
4      if (tax(X,i)+tax(X,j) == S) // (*)
5        maximum = max(maximum, tax(Y,i)+tax(Y,j));
```

なおこの例題はさらに，繰り返し回数を減らすための工夫を検討することもできるが，今回は必須ではない．

■計算量の簡単な検討 (estimation of the number of trials)　上記のプログラムで 4 行目 (*) が最大何回実行されるか，考察しよう (厳密に求めなくて良い)．回数を (S に依存するので)，$f(S)$ と表すとする．2 行目の for ループが S 回，3 行目の for ループが最大 S 回繰り返すので，$f(S) \leq S^2 \leq 1\,000^2 = 10^6$ である．

表 2.2　C++ で，1 秒間で実行可能な演算回数の目安 ([3] を改変して転載)

1 000 000	余裕を持って可能
10 000 000	たぶん可能
100 000 000	とてもシンプルな演算なら可能

この回数 (10^6) を表 2.2 に当てはめると，(tax 関数が効率的に実装されていることを前提に) 余裕を持って，1 秒間で実行可能と分かる．[*2]表の数値は，実行環境に合わせて経験的に測定する必要がある．目安として，最近の多くのコンピュータの CPU は 1GHz (= 10^9) 程度で動作しているので，CPU が 100 サイクル以内に実行できる演算なら，1 秒間に 10^6 回実行可能と期待できる．

本演習の範囲では，この表を信じて問題ない場合がほとんどであるが，現実の計算機は複雑な装置であるから，正確な実行時間の予想は簡単ではない．たとえばメモリ参照や new/delete は，算術演算よりそれぞれ 10 倍，100 倍以上に遅い場合がある．様々な条件が影響するため，予測のためには何らかのキャリブレーションが必要である．また，C, C++ 以外の言語では，実行により時間がかかることも多い．

■複数データセットの入力　この問題ではデータセットが複数与えられうる．すなわち，データが与えられる間はそれを読み込んで解を出力し，データが終了したらプログラムを終了させる必要がある．「入力の終わりは，空白で区切られた 3 つのゼロからなる行によって示される． 」という条件と通常のデータセットでは `X` が正であることを活用し，以下の構造でプログラムを書くことを勧める．

C++

```
#include <iostream>
using namespace std;
int X, Y, S;
int solve() {
  ... // X,Y,S について，最大値を計算
}
int main() {
    while (cin >> X >> Y >> S && X>0) {
        cout << solve() << endl;
    }
}
```

`while` 文の条件の `cin >> X >> Y >> S` の部分は cin への参照を返し，`bool` にキャストされた際に，`cin` が正常状態かどうか，すなわち `X`, `Y`, `S` を正常に読み込めたかどうかを返す．入力ファイルが間違っていた (よくある場合は，タイプミスまたは違う問題の入力を与えた) 場合はここが `false` になって終了する．読み込めた場合に，`X`, `Y`, `S` は処理すべきデータセットの `X`, `Y`, `S` を読んだ場合と，入力終了の目印の空白で区切られた 3 つのゼロを読んだ場合の両方がある．前者の場合は正の整数であるはずなので，0 かどうかをテストして判別する．

■AOJ への提出　上記の方針でプログラムを作成し，AOJ で accept されることを確認せよ．

もし accept が得られなかった場合は手元での検証が可能である．まず，入力 (http://icpc.iisf.

[*2] 注: この見積もりは 1 つのデータセットについてのものである．一方，問題全体では「入力は複数のデータセットからなる」ものを 8 秒間で回答する必要があるため，計算時間はデータセット数の上限を考慮して見積もる必要がある．データセットの数について記述がない場合は，厳密には問題の不備であるが，10 個程度と考えて良い．

or.jp/past-icpc/domestic2014/qualify14_ans/A1) と正答 (http://icpc.iisf.or.jp/past-icpc/domestic2014/qualify14_ans/A1.ans) をダウンロードしておく.

```
$ ./a.out < A1 > my-output.txt
$ diff -u A1.ans my-output.txt
```

差があれば行数が表示される.

2.3 試行回数の見積もりと計算手順の改良

続いて，計算方法の微調整によって試行回数が異なる例題を試してみよう.

問題 **Space Coconut Crab** (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 ← 同じ数値でも重要な制約と余計な情報の区別に注意)

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

考察課題: 下記を参照して，各方針の計算コストに関するグラフと考察を提出する.

まず，変数 x, y, z の組み合わせを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．この問題は最小化したい式が $f(x, y, z) = x + y + z$ と単純な形をしているので試行錯誤なしに解くことができるが，効率的な解放の有無は $f(x, y, z)$ に依存する．)

C++

```
int cnt = 0; // 計測用 global 変数
int solve(int E) {
  // 愚直な求め方
  int ans = ... // 最大値
  for (int x=0; x<=E; ++x)
    for (int y=0; y*y<=E; ++y)
      for (int z=0; z*z*z<=E; ++z) {
        ++cnt;
        if (x+y*y+z*z*z == E)
            ... // x+y+z が最小値なら ans を更新
      }
  return ans;
}
int main() {
  int E;
  while (cin >> E && E>0) {
    cout << solve(E) << endl;
```

```
18    }
19  }
```

前問同様に 8 行目が何回実行されるかを見積もってみよう．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．そのような (x,y,z) の組み合わせは，E の最大値が 1,000,000= 10^6 であることから，最大 $10^6 \cdot 10^3 \cdot 10^2 = 10^{11}$ である．表 2.2 を参照すると，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．

実際に，main を以下のように差し替えると，各 E の値に対応する試行回数を表示することが出来る．

C++

```
int main() {
  int E=1000000;
  solve(E);
  cout << "count = " << cnt << endl;
}
```

まとめて計測する場合はループを作ると良い

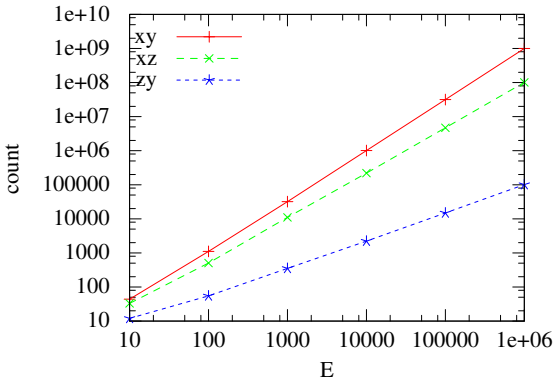
C++

```
int main() {
  int E[]={100,10000,1000000};
  for (int i=0; i<3; ++i) {
    cnt=0;
    solve(E[i]);
    cout << "E = " << E[i] << "count = " << cnt << endl;
  }
}
```

試行回数を減らすために，全ての (x,y,z) の組み合わせではなく，$x + y^2 + z^3 = E$ を満たす範囲のみを考えることにする．たとえば，x と y を決めると，$E - x - y^2 = z^3$ から z を決定できる (z が整数とならない場合，すなわち z=(int)round(pow(E-x-y*y,1.0/3)) として x+y*y+z*z*z==E 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 と y を決めて x を求めることもできる．

一番良い方針が制限時間に間に合う範囲であることを確認し，AOJ に提出して確かめる．AOJ に提出する際は，cnt の表示は消しておくこと．

■考察課題　以下のグラフは，E を 10 から 10^6 まで適当に変えた時に，調べる組み合わせの種類数を実測したものである．たとえば xy は x,y,z の順に決めた場合を表す．



方針 xy と zy のプログラムを書き，いくつかの E に対する試行回数を出力し，グラフの数値と比較せよ．E の大きさと試行回数の関係に関する考察を，提出するソースコード内にコメントとして記入せよ．注意 このような数値実験の際は，ナマの数値を保存しておくこと．また転記の際は，手打ちではなく，コピーペーストを用いること (転記ミスを防ぐため)．プログラムの出力は次のように redirection を用いて，ファイルに保存する．

```
$ ./a.out > xy-e-vs-count.log
```

(オプション:総合情報コースの人はなるべく) グラフを自作して提出する．グラフは，gnuplot[*3] , matplotlib (python) などを勧めるが，R など馴染みのツールがあればそれで良い．

(オプション) 試行回数とプログラムの実行時間の関係を調査せよ．たとえば横軸を試行回数，縦軸を実行時間として散布図を描く．プロセス全体の実行時間の測定は，`time` コマンドが利用可能である．

```
$ time ./a.out
```

関数毎の所要時間を測る方法には，一例として C++11 から標準となった `chrono` ライブラリを紹介する．C++11 の機能を使うには，`g++ -Wall -std=c++11 filename.cc` のようにコンパイルオプションに`-std=c++11` を追加する．

C++11

```
#include <chrono>
using namespace std;
using namespace std::chrono;
  auto t0 = high_resolution_clock::now();
  solve(E);
  auto t1 = high_resolution_clock::now();
  cout << duration_cast<duration<double>>(t1-t0).count()
```

[*3] https://hwb.ecc.u-tokyo.ac.jp/current/applications/gnuplot/

```
8         << "␣seconds\n";
```

いずれの場合も，様々な理由で実行時間は試行毎に変化しうることに注意．

答えが合わない場合の参考

変数 y，z などの for 文を書くときには，範囲に注意する．0 を含むか，上限は十分か．浮動小数誤差の影響で，pow(a,1.0/3) などの値は実際よりも小さくなりうる．一般に，z <= pow(a,1.0/3) という式の代わりに，z*z*z <= a と判定するほうが良い．

2.4 計算量の節約

問題 **Square Route**⋆ (模擬国内予選 2007)

縦横 1500 本程度の道路がある街で，正方形の数を数えてほしい．http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2015

考察課題: 手法の説明と工夫した点，オーダを説明せよ．(考察が十分かどうか，なるべく授業中に，確認を受けること)

全ての 4 点を列挙して正方形になっているかどうかを試すと，$O(N^2M^2)$ となって間に合わない．様々な工夫の余地がある．ヒント:

問題 **ほそながいところ** ⋆ (夏合宿 2012)

馬車 n 台の出発時刻を調整して，条件を満たしながら全ての馬車がゴール地点に到着するまでにかかる時間の最小値を求める．

補足: この問題は難しいので，全部でのど可能性を試せばよいか，どのようにフィルタするか，解を求めるにはどのような関数を作成すれば良いか，方針をなるべく具体的に記述すれば評価する

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2427

2.5 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことなど) とともに，ITC-LMS に提出する．コメントの記述は，#if 0 と#endif で囲むことが簡便である．

C++

```
1  // minmaxsum.cc
2  #if 0
3    所要時間は..であった. ...を復習した.
4    ...
```

```
5  #endif
6  int main() {
7  }
```

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを，提出する．内容はソースコードとコメントの文章とする．(プレーンテキストであること．つまり，PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，別ファイルで提出する．
  ⋆ つきの問題はオプションである．なお ⋆ つきの問題に (教員の基準で) 完全な回答の提出を行った場合は，⋆ なしの問題の提出を免除する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

2.A 型パラメータ ⋆

(発展) 型を取り替えながら似たような処理を行いたい場合がある．C++ では `template` 関数という仕組みを用いることで，型以外の共通部分を関数としてまとめることができる．

C++

```
1  template <class T>
2  void show() { // 型 T の情報を表示
3    cout << sizeof(T)
4         << '␣' << numeric_limits<T>::digits
5         << '␣' << numeric_limits<T>::digits10
6         << '␣' << numeric_limits<T>::min()
7         << '␣' << numeric_limits<T>::max()
8         << endl;
9  }
10 int main() {
11   show<int>(); // T=int として show を呼ぶ
12   show<long>();// T=long として show を呼ぶ
13   show<long long>();
14   show<float>();
15   show<double>();
16   show<long double>();
17 }
```

第 3 章

整列と貪欲法

2018-10-10 Wed

概要

ある基準に従ってデータを並べ替えることを整列(sort) という．ほとんどの言語の標準ライブラリでは，整列の方法が提供されているので，まずはそれの使い方を習得しよう．この資料では整列された結果で得られれば良いという立場をとるが，整列の手法そのものに興味がある場合は，アルゴリズムの教科書を参照されたい．
データの整列は，問題解決の道具として活躍する場合もある．それらには，最適化問題や配置問題など，整列とは関係がない見た目の問題も含まれる．

3.1　数値と文字列の整列 (sort)

C++ と Python では，標準関数として `sort` が用意されている．整列に要する計算量は列の長さを N として $O(N \log N)$ である．[*1] 通常は，N に対して $\log N$ は小さいため，整列の計算量が問題になることは少ない．

C++

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
int A[5] = {3,5,1,2,4};
int main() {
    sort(A,A+5); // 半開区間 [l,r) で指定する．sort(&A[0], &A[5]) と同じ意味
    ... // cout に A を出力してみよう
}
```

配列を整列させる範囲を指定するには，`&A[0]` のように配列の要素を指すポインタを用いる．一次元配列の場合に単に `A` と書けば，先頭要素を指すポインタに自動的に変換される．すなわち，`sort(A,A+5)` は `sort(&A[0], &A[5])` と同じ意味であり，`A[0]` から `A[4]` まで (端点含む) が整列される．`sort(A,A+3);` などと，一部の区間のみを指定して整列することもできる．

Python3

[*1] ただし実装とデータ次第で $O(N^2)$ かかるものがあることに注意．

```
a = [3,5,1,2,4]
a.sort()
a
# [1, 2, 3, 4, 5]
```

例題 **Sort II** (AOJ)

与えられた n 個の数字を昇順に並び替えて出力するプログラムを作成せよ.
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=10029

回答例:

C++

```
int N, A[1000000+10];
int main() {
    cin >> N;
    for (int i=0; i<N; ++i) cin >> A[i]; // A の入力
    ... // A をソートする
    for (int i=0; i<N; ++i)
        cout << (i?" ":"") << A[i]; // A の出力
    cout << endl;
}
```

■文字列の整列 標準ライブラリの sort 関数は，数値だけでなく文字列string の整列を行うことも出来る．一般に，比較演算子<が定義されている要素の整列が可能である．

例題 **Finding Minimum String** (AOJ)

N 個の小文字のアルファベットのみからなる文字列の，辞書順で先頭の文字列を求める．問題文中に記述がないが N は 1000 を越えない．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=10021

補足: 辞書式順序は，大文字小文字が混ざる場合は C++ の std::string の比較演算子の比較順序とは異なる．(が，今回は小文字のみなのでその心配はない)

C++

```
#include <algorithm> // sort のため
#include <string> // string (文字列) のため
#include <iostream>
using namespace std;
string A[1000];
int N;
int main() {
    cin >> N;
```

```
9       if (N > 1000) abort();
10      for (int i=0; i<N; ++i) cin >> A[i];
11      ... // 整数の時と同様に A を整列 (sort) する
12      cout << A[0] << endl;
13  }
```

3.2 ペアと整列 (pair)

3.2.1 ペア

ペア(組み) の表現を導入する．⟨ 開始時刻, 終了時刻 ⟩ や ⟨ 身長, 体重 ⟩，⟨ 学籍番号, 点数 ⟩ など，現実世界の情報にはペアとして表現することが自然なものも多い．

整数のペアは以下のように構造体を用いて表現できる．

C++

```
1  struct pair { // 注： 下記の説明に従えば，この入力は不要
2      int first;
3      int second;
4  };
```

同様の使い勝手のものpair が標準ライブラリに用意されていて，こちらの利用を勧める．整数のペアは pair<int,int>で表現され，int を他の型に変えれば他のペアも利用可能である．ペアの 2 つの要素を同時に初期化する略記法として，make_pair が用意されている．

C++

```
1  #include <utility> // pair のため
2  #include <iostream>
3  using namespace std;
4  int main() {
5    pair<int,int> a(2,4); // 整数のペア
6    cout << a.first << '␣' << a.second << endl; // 2 4 と表示
7
8    a.first = 3;
9    a.second = 5;
10   cout << a.first << '␣' << a.second << endl; // 3 5 と表示
11
12   a = make_pair(10, -30);
13   cout << a.first << '␣' << a.second << endl; // 10 -30 と表示
14
15   pair<double,char> b; // 小数と文字のペア
16   b.first = 0.5;
17   b.second = 'X';
18   cout << b.first << '␣' << b.second << endl; // 0.5 X と表示
19 }
```

3.2.2 ペアの配列と整列

続いて，ペアの配列を扱う．たとえば，一人の身長と体重をペアで表現する時に，複数の人の身長と体重のデータはペアの配列と対応づけることができる．前回，整数の配列を整列 (sort) したように，ペアの配列も整列することができる．標準では，第一要素が異なれば第一要素で順序が決まり，第一要素が同じ時には第二要素が比較される．

C++

```cpp
#include <utility> // pair のため
#include <algorithm> // sort のため
#include <iostream>
using namespace std;
int main() {
  pair<int,int> a[3]; // 整数のペア
  a[0] = make_pair(170,60);
  a[1] = make_pair(180,90);
  a[2] = make_pair(170,65);

  for (int i=0; i<3; ++i) // a[0] から a[2] まで表示
    cout << a[i].first << ' ' << a[i].second << endl;
  // 170 60
  // 180 90
  // 170 65 と表示されるはず

  sort(a, a+3); // a[0] から a[2] まで整列

  for (int i=0; i<3; ++i) // a[0] から a[2] まで表示
    cout << a[i].first << ' ' << a[i].second << endl;
  // 170 60
  // 170 65
  // 180 90 と表示されるはず
}
```

Python3

```python
a = [[3,5],[2,9],[3,6]]
print(a) # [[3, 5], [2, 9], [3, 6]]
a.sort()
print(a) # [[2, 9], [3, 5], [3, 6]]
```

3.3 貪欲法 (greedy algorithms)

整列の重要な応用として貪欲法(greedy algorithm) を紹介する．

3.3.1 整数の整列と貪欲な選択

問題	カントリーロード	(UTPC 2008)

カントリーロードと呼ばれるまっすぐな道に沿って，家がまばらに建っている．指定された数までの発電機と電線を使って全ての家に給電したい．家に電気が供給されるにはどれかの発電機に電線を介してつながっていなければならず，電線には長さに比例するコストが発生する．できるだけ電線の長さの総計が短くなるような発電機 および電線の配置を求める．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2104

考察課題 なぜ正しい回答を得られるかの説明をまとめよ

ヒント: 発電機が一つなら，端から端まで全ての家を電線でつなぐ必要がある．発電機が 2 つなら，端から端まで全ての家を電線でつないだ状態から，家と家の間の一箇所に電線を引かずに節約することができる．つまり，一番長い一箇所を節約するのが得．発電機が 3 つなら，端から端まで全ての家を電線でつないだ状態から，家と家の間で一番長い部分と二番目に長い部分には電線を引かずに節約することができる．...

方針の確認

プログラムを書く前に Sample Input を手で解いてみよう

回答例

C++

```
1  int T, N, K, X[100000+10], A[100000+10];
2  int main() {
3      cin >> T; // データセットの数を読み込む
4      for (int t=0; t<T; ++t) {
5          ... // 家と発電機の数を読み込む
6          for (int i=0; i<N; ++i) cin >> X[i]; // 家の位置を入力
7          for (int i=0; i+1<N; ++i) A[i] = X[i+1]-X[i]; // 家と家の間
8          ... // 配列 A を整列する
9          ... // 配列 A の先頭 max(0,N-1-(K-1)) 個の和を出力する
10     }
11 }
```

よくある考え漏れ

発電機 節約できる区間は増えない (ヒントが白い文字で書かれているので，読みたくなったらコピーペーストなどで読むと良い).

このような解法が貪欲法 (greedy) と呼ばれる意味は，節約する区間の「組み合わせ」を考えることなく，(長さ順に並べた) 単独の区間を順に一つづつ見て閾値を越えるまで節約することを貪欲に決める点である．

3.3.2 ペアの整列と選択

問題	**Princess's Marriage**	(模擬国内予選 2008)

護衛を上手に雇って，道中に襲われる人数の期待値を最小化する．護衛は 1 単位距離あたり 1 の金額で，予算がある限り，自由に雇える．

入力は，区間の数 N, 予算 M に続いて，距離と 1 単位距離あたりの襲撃回数の期待値のペア ⟨D,P⟩ が N 個.

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2019

考察課題 問題の入力と計算量のオーダに関して説明せよ

考え方: せっかく護衛を雇うなら，もっとも危険な区間を守ってもらうのが良い．一番危険な道を選び，予算がある限りそこに護衛を雇う．予算の残額で道の区間全てをカバーできない場合は，安全になる区間と，危険なままの区間が生ずる．予算が残っている限り，2 番目に危険な道，3 番目に危険な道の順に同様に繰り返す．残った危険な区間について，期待値の和が答えとなる．この計算課程では，⟨ 危険度, 長さ ⟩ のペアで道を表現し，危険な順に整列しておくと便利である. (ここで危険度は，距離 1 移動する間に襲われる回数の期待値を表す)

方針の確認

プログラムを書く前に Sample Input を手で解いてみよう

回答例 (入力と整列):

C++

```
int N, M;
pair<int,int> PD[10010];
int main() {
  while (cin >> N >> M && N) {
    int d, p;
    for (int i=0; i<N; ++i) {
      cin >> d >> p;
      PD[i] = make_pair(p, d);
      // PD[i].first は道 i の危険度
      // PD[i].second は道 i の長さ
    }
    ... // PD を大きい順に整列しよう
    // 整列がうまく行ったか，PD を表示してみよう
    // うまく整列できたら，次は答えを計算しよう
  }
}
```

回答例 (答えの計算):

C++

```
    int S = 0;
```

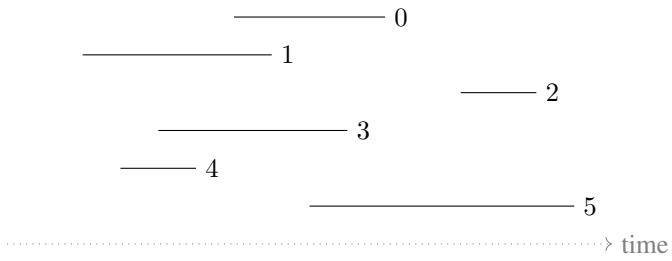
```
2       for (int i=0; i<N; ++i)
3         S += 道[i]の危険度 * 道[i]の長さ;
4       // 予算 0 の時の答えが，現在の S の値
5       for (int i=0; i<N; ++i) {
6         if (M <= 0) break;
7         int guarded = Mと道[i]の長さの小さい方; // 扉う区間
8         S -= 道[i]の危険度 * guarded;
9         M -= guarded;
10      }
11      S が答え
```

- 予算が 0 の場合は，答えとして必要な “刺客に襲われる回数の期待値” である S は，各道 i について $S = \sum_i P_i \cdot D_i$ となる．
- 予算が M の場合，S から護衛を雇えた区間だけ期待値を減らす．例えば道 j の区間全部で護衛を雇うなら $P_j \cdot D_j$ だけ S を減らすことができる．もし残り予算 m が道の長さ D_j より小さく道の一部だけしか護衛を雇えないなら S から減らせるのは $P_j \cdot m$ だけである．

3.3.3 区間スケジューリング

「アルゴリズムデザイン」[4] の 4.1 章 (pp. 104–108) を参照.

■問題概要 一つの共有資源 (体育館，駐車場など) と多数の利用申請 (開始時刻と終了時刻のペア) があったとする．使用時間が衝突しないように，利用申請の部分集合を選ぶ．最大いくつの申請を採用できるか?



単純化: 体育館や時刻などの具体を削ぎ落とすと，線分の集合から一部を選択する問題になる．

■直感と考察

- 実は左から採用すれば良いのでは? ➔ そうでもない (反例有り)
- 短い線分は他と重なりにくい．短い線分から採用すれば良いのでは? ➔ 必ずしもそうでもない (反例有り)

■正しいアルゴリズム

1. 右端が一番左にある線分(i.e., 早く終わる申請) を選び，重なる線分とともに取り除く
2. 未選択の線分がまだあれば，step.1 に戻り手順を繰り返す

問題	**Radar Installation**	(Beijing 2002)

全ての島が見えるようにレーダーを配置する際に、レーダーの数を最小化したい
http://poj.org/problem?id=1328
注意: y=0 の島が存在する模様，x の絶対値が 2000 以下であることは保証される．
考察課題 なぜ正しい回答を得られるかの説明を述べよ

POJ のアカウントは、AOJ とは別に作る必要がある。
ヒント:

- 島が観測できるレーダーの区間を列挙したとする．まだレーダにカバーされていない島を一番沢山カバーできる場所にレーダを配置し，それを繰り返すという戦略を考える．この戦略が最適な配置より多くのレーダを必要とするような島の並びの例を示せ
- 左から順にレーダを配置するとする．(まだレーダにカバーされていないなかで) 見えはじめるのが最も左にある島を選び，その島が見える区間の左端にレーダを配置するとする．この戦略が最適な配置より多くのレーダを必要とするような島の並びの例を示せ
- 左から順にレーダを配置するとする．(まだレーダにカバーされていないなかで) 見えはじめるのが最も左にある島を選び，その島が見える区間の右端にレーダを配置するとする．この戦略が最適な配置より多くのレーダを必要とするような島の並びの例を示せ

厳密に説明を行う場合は、レーダを配置する正しい戦略 A を示し，他のどのような配置も戦略 A による配置よりレーダ数が小さくならないことを背理法で証明する。

よくあるバグ

遠すぎる島など，例外を考慮する必要がある．複数のテストケースを扱うので，遠すぎる島があっても入力は全ての島を読み込むこと (さもないと次のテストケースで先頭がずれる).

問題	**Dinner***	(夏合宿 2004)

自炊か外食かを考えながら、N 日間で得られる幸福度の合計を最大化する。
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2642
考察課題 なぜ正しい回答を得られるかの説明をまとめよ

ヒント: k 回自炊するとしたら，どの日に行うか良いかを考えてみよう．さらに，

3.4 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する．

コメントの記述は，#if 0 と#endif で囲むことが簡便である．

C++

```cpp
// minmaxsum.cc
#if 0
   所要時間は..であった．方針は....
   ...
#endif
int main() {
}
```

扱う問題は以下の箇条書きのどちらかで合格 (言語指定なし):

- 「カントリーロード」と (「Princess's Marriage」または「Radar Installation」) を解いて AC を得て考察とともに提出．(Rader Installation の方を勧める)
- 「Dinner」で AC を得たうえで考え方の説明とともに提出する (どの問題も解いたことがある場合は，「似た」方針と難易度の問題を一つ探して，回答・解説し，提出する)

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

3.A 列に対する操作

3.A.1 反転 reverse

C++

```cpp
#include <algorithm>
using namespace std;
int A[5] = {3,5,1,2,4};
int main() {
    sort(A,A+5);
    reverse(A,A+5); // 与えられた範囲を逆順に並び替え
    ... // cout に A を出力してみよう
}
```

Python3

```python
a = [3,5,1,2,4]
a.sort()
a.reverse() # a を逆順に並び替え
```

3.A.2 回転 rotate

C++ で `rotate(a,b,c)` は範囲 [a,b) と範囲 [b,c) を入れ替える．

C++

```cpp
#include <algorithm>
int A[7] = {0,1,2,3,4,5,6};
int main() {
  rotate(A,A+3,A+7);
  // vector の場合は rotate(A.begin(),A.begin()+3,A.begin()+7);
  // A を出力してみよう ➔  3 4 5 6 0 1 2
}
```

Python では，`a[p:q]` でリスト a の位置 p から q の要素を参照したり，置き換えることができる．

Python3

```python
a = [0,1,2,3,4,5,6]
a[0:7] = a[3:7]+a[0:3]
print(a) # ➔  [3, 4, 5, 6, 0, 1, 2]
```

3.A.3 大きい順に整列

標準の `sort` 関数は小さい順に並べ替える．大きい順に並べ替えるにはどのようにすれば良いだろうか?

1. 小さい順に並べ替えた後に，並び順を逆転させる (当面これで十分)

C++

```cpp
int A[5] = {3,5,1,2,4};
int main() {
   sort(A,A+5);
   reverse(A,A+5); // 与えられた範囲を逆順に並び替え
   ...  // cout に A を出力してみよう
}
```

Python3

```python
a = [3,5,1,2,4]
a.sort()
a.reverse() # a を逆順に並び替え
```

2. `reverse_iterator` を使う

C++14

```cpp
  sort(rbegin(A),rend(A));
```

Python3

```python
a.sort(reverse=True)
```

3. 比較関数を渡す

C++11

```
1 int A[5] = {3,5,1,2,4};
2 int main() {
3    sort(A,A+5,[](int p, int q){ return p > q; });
4    ...  // cout に A[i] を出力してみよう
5 }
```

文法の説明は省略するが `[](int p, int q){ return p > q; }` の部分が 2 つの整数の並び順を判定する匿名関数である．

Python3

```
1 a.sort(key=lambda e: -e)
```

第 4 章

文字列と標準データ構造, 区間の調査

2018-10-17 Wed

お知らせ

poj.org や szkopul.edu.pl のアカウントは aoj とは別なので、適宜作成してください。なお，どのサイトも好意で運営されているので，迷惑をかけないこと (パスワードをなくさないこと).

概要

スタック，キュー，優先度付きキュー，連想配列などのライブラリの利用法を習得し，必要に応じて応用できるようになる．データ構造を自作する必要はない．

4.1 文字列 (string) と入出力

C++

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main() {
5   string word; // 文字列型の変数 word を定義
6   cin >> word; // 標準入力から一単語読み込む (空白や改行文字で区切られる)
7   cout << word << word << endl; // 2 回表示する
8 }
```

hello(改行) と入力すると hellohello と出力される．

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．

C++

```
1 #include <string>
2   string word; // 定義
3   string word2="ABCD"; // 定義と同時に初期化
4   word = "EF"; // 代入
5   string word3 = word + word2; // 連結
6   char c = word[n]; // n 文字目を取り出す
7   word[n] = 'K'; // n 文字目に代入 (n<word.size() でないと破綻)
8   cin >> word; // 一単語読み込み (改行や空白文字で分割される)
9   cout << word.size() << endl; // 文字数表示
```

```
10    if (word.find("A") != string::npos) ... // もし word に A が含まれるなら...
```

Python3

```
1  word = input().strip()
2  print(word*2)
```

1 単語読み込むことと 1 行読み込むことは意味が異なるが，今回は違いに踏み込まない．

4.2　スタック (stack) とキュー (queue)

スタックとキューは，データを一時的に保存したり取り出したりするためのデータ構造である．どちらもデータを 1 列に並べて管理して，新しいデータを列の (どちらかの) 端に保存し，また取り出す際も (どちらかの) 端から取り出す．

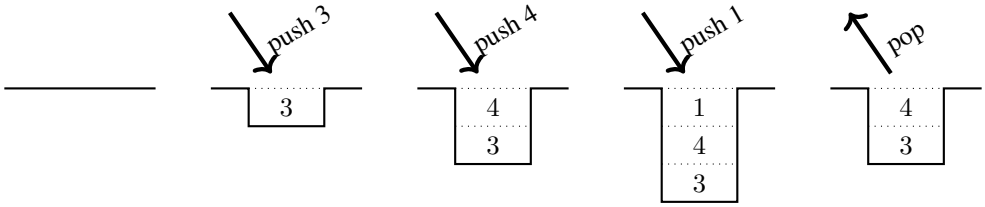
スタックは，後に保存したデータを，先に取り出すデータ構造である．たとえば，仕事をしている最中に急な案件が発生したので，今取り組んでいる仕事を「仕事の山」の先頭に積んでおいて，急な案件を処理し，それが終わったら仕事の山の先頭から再び処理を続けるような状況で有用である．もちろん急な案件の処理中に，さらに急な案件が発生した場合も同様に仕事の山に積んだり下ろしたりする．

キューは，先に保存したデータを，先に取り出すデータ構造である．飲食店に到着した人は列の後ろに並び，列の中で一番早く到着した人から順に店内に案内される状況に相当する．

優先度付きキューでは，早く並んだ順ではなく各データの優先度の高い順に取り出される．現実にはぴったりした例がないが，待っている人の中から優先度の高い順に診療が行われるような状況や，高い買値を提示した順に品物を入手できる市場などがもしあれば相当するだろう．

4.2.1　スタック

■概要　スタック (stack) は，`push()` と `pop()` という二つの操作を提供するデータ構造である．それぞれ，データの追加と，取り出しを行う．現在先頭にあるデータは，`top()` で参照できる．



■使用上の注意　通常は，配列や `vector` などが用いられる．計算コストは `push()`，`pop()`，`top()` とも定数時間である．つまり，データがどれだけ増えても，計算時間は増えないと期待される．

なお，`pop()` が `top()` 同様に値を返す実装もあるが，C++ の標準ライブラリで提供されているものはでは，両者が分離されていることに注意[*1]．通常の使用状況では，先頭の要素を得つつスタックやキューからは取り除きたいので，両者の操作を続けて行う．

[*1] これは exception safety のためである．

また現在の要素数を調べる size() などのメンバ関数も提供されているので詳しくは文法書を参照のこと[*2].

C++

```cpp
#include <stack>
#include <iostream>
using namespace std;
int main() {
    stack<int> S; // int を格納するスタック
    S.push(3); // 先頭に追加
    S.push(4);
    S.push(1);
    while (! S.empty()) { // 要素がある間
        int n = S.top(); // 先頭をコピーして
        S.pop(); // 先頭要素を廃棄
        cout << n << endl; // 1, 4, 3 の順に表示される
    }
}
```

Python では append, pop をそれぞれ push, pop として用いることにする.

Python3

```python
stack = []
stack.append(3)
stack.append(4)
stack.append(1)
while len(stack) > 0:
    n = stack.pop()
    print(n)
```

例題　**Reverse Polish Notation**　(AOJ)

「逆ポーランド記法」で与えられる文法を読んで、値を計算する
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_3_A&lang=jp

ヒント: 問題分末尾にある「解説」も参照.

C++

```cpp
#include <string>
#include <stack>
#include <iostream>
using namespace std;
int main() {
  string word;
  stack<int> S;
```

[*2] 非公式ではあるがウェブにも資料がある http://www.cplusplus.com/reference/stack/stack/

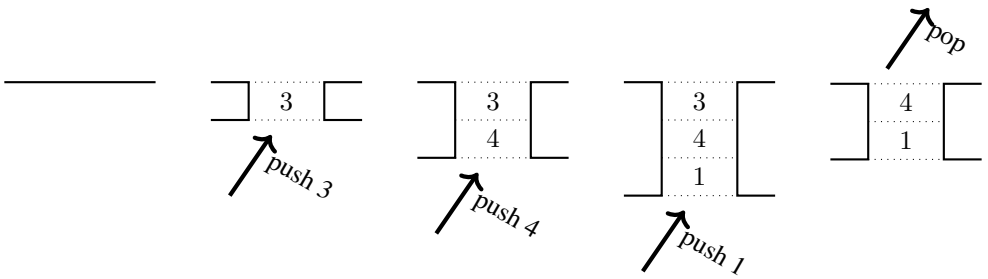
```
8    while (cin >> word) { // 入力がある限り読み込む
9      if (word == "+") {
10       // 数を 2 つ pop して、和を push する
11     }
12     else if (word == "-") {
13       // 数を 2 つ pop して、差を push する
14     }
15     else if (word == "*") {
16       // 数を 2 つ pop して、積を push する
17     }
18     else {
19       // word を数値にして push する
20     }
21   }
22   // S の先頭要素を表示する。
23 }
```

このプログラムは入力が続くかぎり読み込む。キーボードから入力の終わりを与えるには “^D” (Ctrl キーを押しながら “d” を押す) を用いる。

数をあらわす文字列を整数に変換するには、C++11 の場合は `stoi(word)` を、そうでない場合は `<cstdlib>` を include して、`atoi(word.c_str())` などとする。

4.2.2 キュー

■概要　キュー (queue) は，データの格納 (`push()`) と取り出し (`pop()`) ができるデータ構造である．現在先頭にあるデータは，`front()` で参照する．



(empty)　front= 3　front= 3　front= 3　front= 4

■使用上の注意　実装には，配列や `deque` が用いられる．計算コストは `push()`，`pop()`，`front()` とも定数時間である．つまり，データがどれだけ増えても，計算時間は増えないと期待される．

C++ では，queue というテンプレートクラスが用意されている．入れた (push した) データを，front() 及び pop() の操作により取り出す．

C++

```
1  #include <queue>
2  #include <iostream>
3  using namespace std;
4  int main() {
5      queue<int> Q; // int を格納するキュー
```

```
6       Q.push(3); // 末尾に追加
7       Q.push(4);
8       Q.push(1);
9       while (! Q.empty()) { // 要素がある間
10          int n = Q.front(); // 先頭をコピーして
11          Q.pop(); // 先頭要素を廃棄
12          cout << n << endl; // 3, 4, 1 の順に表示される
13      }
14  }
```

他のメンバ関数について詳しくは文法書を参照のこと[*3].

Python の queue の表現としては，この資料では collections.deque の append, popleft を用いることにする.[*4]

Python3

```
1  import collections
2  q = collections.deque()
3  q.append(3)
4  q.append(4)
5  q.append(1)
6  while len(q) > 0:
7      n = q.popleft() # 先頭から取り出す
8      print(n)
```

例題　Round-Robin Scheduling　(AOJ)

複数の計算を少しづつ処理する様子をシミュレーションしてみよう。
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_3_B&lang=jp

ヒント: 問題分末尾にある「解説」も参照.

4.2.3　優先度付きキュー

優先度付きキュー (priority queue) は，データの格納と取り出しができるデータ構造で，まだ取り出されていない中で大きな順に取り出されるものである.

実装には，二分ヒープなどが用いられる．計算コストは top() とも定数時間であるが，push() または pop() の操作については格納されているデータの数を N として，$O(\log N)$．つまり，データが増えると計算時間が増加することに注意.

C++

```
1  #include <queue>
2  #include <iostream>
```

[*3] 非公式ではあるがウェブにも資料がある http://www.cplusplus.com/reference/queue/queue/
[*4] https://docs.python.jp/3/library/collections.html#deque-objects

```cpp
using namespace std;
int main() {
    priority_queue<int> Q; // int を格納する優先度つきキュー
    Q.push(3); // 追加
    Q.push(4);
    Q.push(1);
    while (! Q.empty()) { // 要素がある間
        int n = Q.top(); // 先頭をコピーして
        Q.pop(); // 先頭要素を廃棄
        cout << n << endl; // 4, 3, 1 の順に表示される
    }
}
```

Python には heapq パッケージがあるので，それを使うことにする．

Python3

```python
import heapq

Q = []
heapq.heappush(Q, 3)
heapq.heappush(Q, 4)
heapq.heappush(Q, 1)
while len(Q) > 0:
    n = heapq.heappop(Q)
    print(n)                    # 1,3,4 の順に表示される
```

例題　**Priority Queue**　(AOJ)

整数を入力として受け入れ、大きい順に取り出すことができる、優先度付きキュー (priority queue) を作成せよ。

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C

STL の priority_queue を用いて，accepted を得られることを確認しよう．

C++

```cpp
#include <queue>
#include <string>
#include <iostream>
using namespace std;
int main() {
  string cmd; // while の外側で定義して一つの変数を再利用する
  priority_queue<int> Q;
  while (cin >> cmd && cmd != "end") {
    if (cmd == "insert") {
       // 何かする
    } else if (cmd == "extract") {
       // 何かする
```

```
13        }
14    }
15 }
```

自分で優先度つきキューを実装する場合は，配列または vector 上に二分ヒープ (binary heap) を作成することが簡便．

4.3 集合と連想配列

4.3.1 集合 (set)

C++ で集合を表現するために標準ライブラリに set 及び unordered_set (C++11 以降) というデータ構造が標準で用意されている．まずは整数の集合の例を紹介する．どちらもほぼ同じ操作を提供するが，ここでは挿入 (insert)，全体の要素数 (size)，指定した要素の有無の検索 (count) を使用する．

実装において set は前者がデータに順序をつけて，unordered_set は順序を無視して管理する．前者は通常二分探索木で実装され，挿入や検索に $O(\log N)$ の時間が必要である．すなわち，データ数 N とともに一回の操作は遅くなることに注意．後者は通常ハッシュ表を使って実装され，各操作は平均的には定数時間でおさまると期待される．

他のメンバ関数について詳しくは文法書を参照のこと[*5]．

C++

```
1  #include <set>
2      typedef set<int> set_t;
3      set_t A;
4      cout << A.size() << endl; // 初めは空なので 0
5      cout << A.count(3) << endl; // 3 は含まれていないので 0
6      A.insert(1);
7      A.insert(3);
8      A.insert(5);
9      A.insert(3);
10     cout << A.size() << endl; // 1,3,5 の 3 つ
11     cout << A.count(3) << endl; // 1 個 (set の場合は最大 1 個)
12     cout << A.count(4) << endl; // 0 個
```

文字列の集合も同様に使うことができる．

C++

```
1  #include <set>
2  #include <string>
3  int main() {
4      typedef set<string> set_t;
5      set_t A;
6      cout << A.size() << endl; // 初めは空なので 0
7      cout << A.count("hello") << endl; // "hello"は含まれていないので 0
8      A.insert("hello");
```

[*5] 非公式ではあるがウェブにも資料がある http://www.cplusplus.com/reference/set/set/, http://www.cplusplus.com/reference/unordered_set/unordered_set/

```
 9      A.insert("world");
10      A.insert("good_morning");
11      A.insert("world");
12      cout << A.size() << endl; // "hello", "good morning", "world"
13      cout << A.count("world") << endl; // 1個 set の場合は最大 1
14      cout << A.count("hello!!!") << endl; // 0個
15  }
```

python の場合は set が用意されている.[*6]

Python3

```
1  s = set()
2  print('a' in s)                # False
3  s.add('a')
4  print('a' in s)                # True
5  s.discard('a')
6  print('a' in s)                # False
```

例題　**Search - Dictionary**　(AOJ)

単語が辞書にある単語かどうかを判定せよ
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_4_C&lang=jp

Time Limit Exceeded?

Time Limit Exceeded (TLE) という判定で AC を得られなかった場合，それは実行時間が許容範囲を越えたことを意味する．C++ で set を使っている場合は，unordered_set を使ってみよう．その場合，提出ウィンドウで言語を C++ ではなく C++11 を選択する．

要素の重複を許す集合 (multiset)

C++ で要素の重複を許す集合を表現するために標準ライブラリに multiset 及び unordered_multiset(C++11 以降) というデータ構造が標準が用意されている．利用方法は，ほぼ同じである．

4.3.2　連想配列

個人番号に対する名前を管理する場合には，配列を用いることが一般的である．

C++

```
1  string name[30];
2  name[0] = "kaneko";
3  name[1] = "fukuda";
```

では，名前に対する個人番号を管理するにはどうすれば良いか．

[*6] https://docs.python.jp/3/library/stdtypes.html#set

C++
```
1 number["kaneko"] = 0; // こんなふうに書きたい
```

連想配列とは，key に対する value を管理する抽象データ型で，上記のような機能を提供する．C++ では map と unordered_map が標準で用意されている．

メンバ関数について詳しくは文法書を参照のこと[*7]．

文字列に対応する数字

C++
```
1  #include <iostream>
2  #include <string>
3  #include <map>
4  using namespace std;
5  int main() {
6    map<string,int> table; // 文字列を数値に対応させる map
7    table["taro"] = 180;
8    table["hanako"] = 160;
9    cout << table["taro"] << endl; // 180
10   cout << table["ichirou"] << endl; // 0
11 }
```

このような例では，key が文字列，value が整数である．

Python3
```
1  all = {}
2  print(len(all))                  # 0
3  all["hello"] = 1
4  all["world"] = 100
5  all["good morning"] = 20
6  print(len(all))                  # 3
7  for k,v in sorted(all.items()):
8      print("%s %d" % (k, v))
9  # good morning 20
10 # hello 1
11 # world 100
```

文字列の出現回数

key が存在しない場合の振る舞いは，処理系と設定に依存するが，上記の例では 0 になるように用いている．これにより出現回数の計測を簡単に行うことができる．

C++
```
1   table["ichirou"]+=1;
2   cout << table["ichirou"] << endl; // 1
3   table["ichirou"]+=1;
4   cout << table["ichirou"] << endl; // 2
```

[*7] 非公式ではあるがウェブにも資料がある http://www.cplusplus.com/reference/map/map/, http://www.cplusplus.com/reference/unordered_map/unordered_map/

Python3

```
# 普通の hash の利用
table = {}
# table["ichirou"] += 1 ...  ng!
table["ichirou"] = table.get("ichirou", 0) + 1
if not "world" in table:
    table["world"] = 1
# Counter の利用
import collections
c = collections.Counter()
c["ichirou"] += 1 # ok
```

出現頻度

例題 **English Sentence** (PC 甲子園 2004)

与えられた文字列について，出現頻度が最も高い単語と，最も文字数が多い単語を出力する
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0029&lang=jp

回答例

C++

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
map<string,int> table;
int main() {
    string /*入力用*/word, /*最大頻度*/frequent, /*最長*/longest;
    size_t N=0; /*文字列frequentの頻度*/
    while (cin >> word) {
        // table[word] を一つ増やす
        // table[word] が N より大きければ，N と frequent を更新
        // word.size() > longest.size() なら longest を更新
    }
    //frequent と longest を出力
}
```

string 型の文字列 s の長さは s.size() で入手することができる.[*8]

[*8] http://www.cplusplus.com/reference/string/string/

4.4 各データ構造を使う問題

問題	**Subsequence**	(Southeastern Europe 2006)

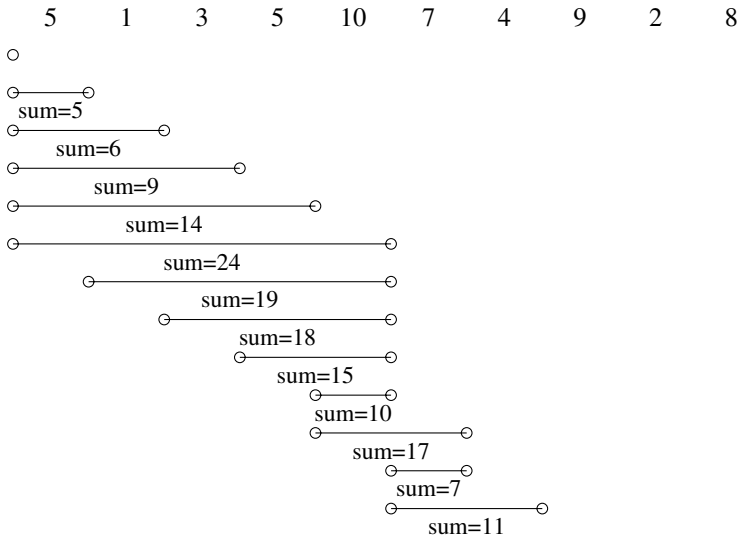
配列 A の連続する部分列について，その和が S 以上となるもののので最小の長さを求めよ．

http://poj.org/problem?id=3061

サンプルの解説: (10,7) の部分が合計 $17 \geq 15$ を満たし要素数 2 で最小．(3,4,5) の部分が $12 \geq 11$ を満たし要素数 3 で最小．

ヒント: 全ての区間を調べると $O(N^2)$ かかるが，次のように合計が S に近い区間のみを調べると $O(N)$ で処理することが出来る．配列の先頭要素から順に見て，キュー内部の要素の合計が S 未満なら `push` して要素を増やし，S 以上なら `pop` して減らす．キュー内部の要素の合計は，それを表す変数を設け，`push` や `pop` のタイミングで管理する．

いわゆるしゃくとり法. たぶん，cin だと遅いので，scanf を使う．



問題	**Organize Your Train part II**	(国内予選 2006)

貨物列車の並べ替え．何通りの可能性があるか?

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1142&lang=jp

文字列 s を反転させるには，`#include <algorithm>`としたうえで `reverse(s.begin(), s.end())` を使う．

ヒント: 分割と前後反転の有無を全通り試す．

C++ では string[*9] データの部分文字列を substr メソッド[*10] で得ることができる．

C++

```
1  string word = "hello";
2  for (size_t i=0; i<=word.size(); ++i) { // size() は文字列の長さ
3    string l = word.substr(0,i);// i 文字目より左の文字列
4    string r = word.substr(i); // i 文字目以降の文字列
5    cout << l << '␣' << r << endl;
6  }
```

実行例:

```
 hello
h ello
he llo
hel lo
hell o
hello
```

練習: string word = "hello"; の hello を別の単語にして動かしてみよう．

Python3

```
1  word = "hello";
2  for i in range(len(word)+1):
3      l = word[0:i]
4      r = word[i:]
5      print(l+'␣'+r)
```

連結には，+ オペレータを用いる．C++ も ruby も同じ．

C++

```
1      string a = "AAA";
2      string b = "BBB";
3      string c = a+b; // 連結
4      cout << c << endl; // AAABBB
```

問題　**Areas on the Cross-Section Diagram**　(AOJ)

与えられた地形に雨が降った際に，できる水たまりの面積を出力する．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_3_D

ヒント: 水面が形成されるとしたら，同じ高さの下りと上りの最近ペアである．文字列を左から順に見て，

[*9] http://en.cppreference.com/w/cpp/string/basic_string
[*10] http://en.cppreference.com/w/cpp/string/basic_string/substr

- 下りであれば位置をスタックに入れる
- 上りであれば，スタックの先頭要素である下りと対応させ，スタックから取り除く

とすることで，同じ高さの下りと上りのペアをつくることが出来る．あとは，隠れてしまう水面を取り除くと良い．一例は，開始位置と面積を pair<int,int>で表現して stack で管理すること．

問題	**Largest Rectangle in a Histogram**	(Ulm Local 2003)

ヒストグラム中に含まれる最大の長方形の面積を求めよ．(long long 注意)
http://poj.org/problem?id=2559

スタックをうまく活用すると，ヒストグラムの本数に比例する手間で求められる．
参考: http://algorithms.blog55.fc2.com/blog-entry-132.html

4.5 応用問題

問題	**Sum of Consecutive Prime Numbers**	(アジア地区予選 2005)

与えられた整数を，連続する素数の和として表せる種類を求めよ．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1257

初めにエラトステネスの篩などの手法で，10 000 までの素数を求めておく．あとは問題 “Subsequence” と同じ．

問題	**Building Blocks⋆**	(15th Polish Olympiad in Informatics)

N 本の棒グラフがある．どこか連続する $K(\leq N)$ 本を選んで，同じ高さに揃えたい．高さを 1 増やすのも減らすのも同じコストがかかる．コストの総和の最小とその時の各棒グラフの高さを出力せよ．$1 \leq K \leq N \leq 100\,000$
https://szkopul.edu.pl/problemset/problem/KC7c6nYfAXCbCGszqhIeOGxP/site/

入出力は scanf でなく cin を用いても問題ない．STL のデータ構造の組み合わせで解くことができる．
ヒント: ある区間を選んだ時のコストは，K 本のデータから簡単に計算できる ( ). 左から順に全ての区間を調べると，ほぼ N の区間がある．それぞれの計算で K 回の計算をすると全体が NK となり間に合わない．区間を右に一つずらした時の差分に注目して，区間を移動した時のコストを手早く $O(\log K)$ 程度で求められると，$O(N \log K)$ となり間に合う．

4.6 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する．

コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である．

C++

```
1 // minmaxsum.cc
2 #if 0
3   所要時間は..であった. 方針は....
4   ...
5 #endif
6 int main() {
7 }
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- 例題を全てと，4.4 節の問題から一題以上を解いて提出する．
  それぞれに，感想(学んだことや課題にかけた時間など)を添えること
- 4.5 節の問題から一題以上を解いて提出する．回答方針や感想(学んだことや課題にかけた時間など)を添えること

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される． この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

第 5 章

漸化式と動的計画法，数え上げ

2018-10-24 Wed

お知らせ (再掲)

poj.org や szkopul.edu.pl のアカウントは aoj とは別なので、適宜作成してください。なお，どのサイトも好意で運営されているので，迷惑をかけないこと (パスワードをなくさないこと).

概要

全ての可能性を調べ尽くすことが難しいような問題について，小さな問題を予め解いて解を表に覚えておくなどの整理を適切に行うことで簡単に解けるようになる場合もある．大小の問題の関係を表す式を立てて，それをプログラムにしてみよう．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．

5.1 数を数える

例題 **Fibonacci Number** (AOJ)

N 番目の Fibonacci 数を求めよ。
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_10_A

簡単のため i 番目の Fibonacci 数を F_i と表記すると，漸化式は以下のように定義される ($F_0 = 0$ と定義することも多いが本質ではないので問題文に合わせる):

$$F_i = \begin{cases} 1 & i = 0 \\ 1 & i = 1 \\ F_{i-1} + F_{i-2} & \text{(otherwise)} \end{cases} \tag{5.1}$$

上記の定義をそのまま再帰で定義すると以下のようになるだろう:

Python3

```
1 def fib(n):
2     print("fib",n) # 関数呼び出しの際に引数を表示
```

```
3       if n == 0:
4           return 0
5       elif n == 1:
6           return 1
7       else:
8           return fib(n-2)+fib(n-1)
```

しかし，この方法は計算の重複が多く，効率的でない．改善方法の一つはメモ化により一度計算した答えを記憶しておくことである (プログラミング演習参照).

余談

フィボナッチ聖人の彫像 (Camposanto, Pisa)

今回主として扱う，より素直な方法は，小さいフィボナッチ数から順に計算することである．式 (5.1) から，F_i を求めるには，F_0 から F_{i-1} までの値のみが必要であり，それらがあれば加算 1 回で計算できるという関係がわかる．そこで F_0 から順に計算すれば，各 F_i は加算 1 回で求められる．

C++

```
1  int F[100]; // 必要なだけ
2  int main() {
3    F[0] = 1;
4    F[1] = 1;
5    for (int i=2; i<45; ++i) { // 必要なだけ
6      F[i] = F[i-2]+F[i-1];
7    }
8  }
```

今回の場合は，回答にあたって直接必要なものは N 番目のデータだけだが，一見回り道のようでも N 番目*まで*のデータを全て計算しておくアプローチが有効である．全体として必要な基本演算の回数 (以下，単に計算量と表記する) は $O(N)$ である．なお，繰り返し自乗法を使うことで $O(\log N)$ に改善することもできるので[*1]，興味のある人は 5.A 節を参照されたい．

この節の目標は，式 (5.1) のような漸化式から，対応するプログラムを書けるようになることである．

[*1] 簡単のために，この資料では整数演算のコストを定数と扱っている．しかし，多倍長整数を使う場合は，大きな数の演算は遅くなることに注意.

例題 **Kannondou** (PC 甲子園 2007)

階段を 1 足で 1,2,3 段上がることができる人が，n (< 30) 段登るときの登り方の種類を，全て試したとして何年かかるかを求めよ (詳細は原文参照).

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0168

各段への登り方の数を配列で表現しよう．漸化式は，i 段に居るために A_i 通りの登り方があるとすると．登る前は $A_0 = 1$, 答えは A_n:

$$A_i = \begin{cases} 1 & i = 0 \\ A_{i-1} & i = 1 \\ A_{i-1} + A_{i-2} & i = 2 \\ A_{i-1} + A_{i-2} + A_{i-3} & \text{(otherwise)} \end{cases} \tag{5.2}$$

フィボナッチ数の計算と同様に，N 段に対する答えを求める計算量は $O(N)$ となる．

配列の範囲外アクセスに注意

C や C++ では配列の範囲外，たとえば `A[-1]`，を参照したり書き込んだりしてはいけない (すぐに実行時エラーになるかもしれないし，ならないかもしれないし，もっと困ったことをしでかすかもしれない)． 典型的には，`A[i-2]` にアクセスする文を書いたら，`i`≤ 1 では実行されないことをプログラマが保証する必要がある．

回答例 (入出力)

C++

```
1     while (cin >> N && N)
2         cout << ((A[N]+..)/10+...)/365 << endl;
```

整数除算で切り上げるには，割る前に (除数-1) を足せば良い (整数除算 `(x+9)/10` が，$x \in [0, 9]$ に対してどのような結果になるかを確認しよう).

Python3 では整数除算には`//`を用いる．

Python3

```
1  while True:
2      n = int(input())
3      if n == 0:
4          break
5      print(((A[n]+9)//10+364)//365)
```

答えが合わない場合の参考

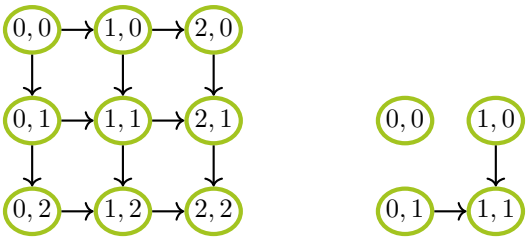
15 段の場合は 5768 通りあり，2 年かかる．

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 (猫にとっての障害物) の落ちている道は通れない．

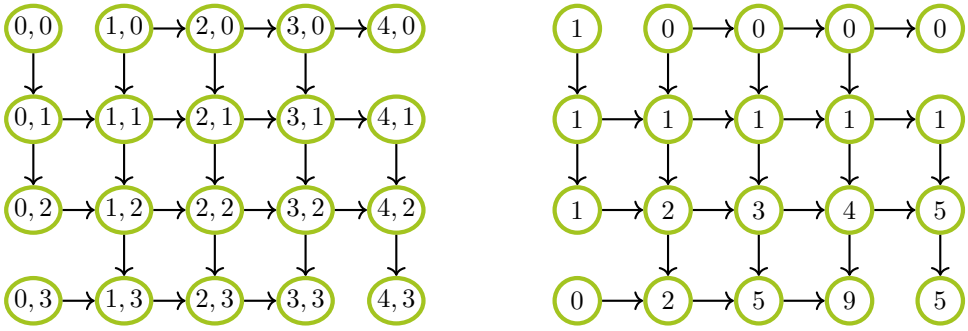
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2186

サンプル入力の初めの 2 つは次の状況を表す。



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，下図左の接続状態を持ち，各交差点は下図右の到達経路数を持つ．



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，上と左からそれぞれ移動可能かどうかを示す配列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

たとえば，ある位置 (x,y) に，上と左のそれぞれの方向からの移動できるかどうかを，`Vert[x][y]` と `Horiz[x][y]` の 2 つの二次元配列で管理する．ある点 (x,y) に上から移動可能であるなら `Vert[x][y]==true`，そうでなければ `Vert[x][y]==false` とする．同様に，左から移動可能かで `Horiz[x][y]` の各要素を設定する．あらかじめ各要素を `true` に初期化しておき，マタタビがあった場合は `false` に書き換える．(移動*不可能*であることを表す場合は `true` と `false` の対応が逆になる．混乱しなければどちらでも良い．)

プログラム作成手順のお勧め

各交差点に到達可能な経路の数を，前ページ図右のように全て表示し，手計算と一致するかどうかを確認しよう．

答えが合わない場合のヒント

またたびの縦横，上端や左端の扱い，$x_1 \leq x_2$ とは限らないこと (y_1, y_2 も) などに注意．

番兵法 (sentinel): 見通しの良いプログラムのヒント

“Kannondou” の例題同様に，`x==0` で `T[x-1][y]` にアクセスしないようにすることと，`y==0` の時 `T[x][y-1]` にアクセスしないようにする必要がある．`if` 文で書くこともできるが，上端と左端に仮想的なマタタビがあると考えて `Vert[x][0]` と `Horiz[0][y]` を設定すると，簡潔に書くことができる．

5.2 さまざまな動的計画法

■最長共通部分列と編集距離　文字列などデータ列から，一部の要素を抜き出して並べた列 (あるいは一部の要素を消して詰めた列) を部分列という．二つのデータ列に対してどちらの部分列にもなっている列を共通部分列という．共通部分列の中で長さが最大の列を最長共通部分列と言う．最長共通部分列は複数存在する場合がある．

問題	**Dynamic Programming - Longest Common Subsequence**	(AOJ)

最長共通部分列の長さを求めよ

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

文字列の長さを N とすると，部分列の候補は 2^N 通りあるので，全て試すと効率が悪い．以下のように工夫して $O(N^2)$ で計算すると良い．

考え方: 文字列 X の先頭 i 文字部分を切り出した部分列を X_i と表す (この式では，文字列の先頭を*1*文字目としていることに注意，多くのプログラミング言語とは 1 ずれている)．X_0 は空文字列とする．文字列 X と文字列 Y の最長共通部分列は，その部分問題である X_i と Y_j の最長共通部分列の長さ $L_{i,j}$ を利用して求める

ことが出来る．

$$L_{i,j} = \begin{cases} 0 & i \leq 0 \text{ または } j \leq 0 \\ 1 + L_{i-1,j-1} & \text{X の i 文字目と Y の j 文字目が同じ文字} \\ \max(L_{i,j-1}, L_{i-1,j}) & \text{otherwise} \end{cases}$$

この計算は二次元配列を利用して，効率よく行うことが出来る．

問題	**Combinatorial - Edit Distance (Levenshtein Distance)**	(AOJ)

二つの文字列の編集距離を求めよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_E&lang=jp

■最適経路を求める　最適値を動的計画法で求めるだけではなく，最適値を実現する具体的なプランを構成したい場合もある．多くの場合には，ほぼ同じ手間で求めることができる．

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

問題	行けるかな?⋆	(UTPC 2008)

サイコロが巨大なすごろくでピッタリとまれる場所を求める (目の合計が 2^{31} まで).
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2107

- 繰り返し自乗法 (5.A 節) を用いる.
- 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.
- ヒント: U ターン禁止に対応するには (以下白い文字で記載)

問題	**Army Training⋆**	(Algorithmic Engagements 2010)

平面上の 1000 個までの点が与えられる．それらを適当につないで単純多角形について聞かれるので，領域に含まれる点の個数を数えよ．(質問数がたくさんあるので，数え方を工夫する)
https://szkopul.edu.pl/problemset/problem/nXF1qOIv5S88utFPI2V_0gt3/site/

ある点が 2 頂点を通る直線の左右にあるか，など基本的な幾何的な計算手法が必要なので，初見の場合は少し後の授業を終えてから取り組むと良い.

5.3 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する.

コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である.

C++

```cpp
// minmaxsum.cc
#if 0
  所要時間は..であった．方針は....
  ...
#endif
int main() {
}
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- 「平安京ウォーキング」を含む問題 2 題以上を解いて提出する.
  それぞれに，感想 (学んだことや課題にかけた時間など)を添えること
- 「行けるかな?」または「Army Training」を含む問題 1 題以上を解いて提出する.
  回答方針や感想 (学んだことや課題にかけた時間など)を添えること

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

5.A　参考: 繰り返し自乗法 (repeated squares)

適切な手法を用いると，計算時間を短縮できることがある．繰り返し自乗法を適用できる場合には，N ステップ後の結果を $\log N$ 回の計算で求めることができる．
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

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

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

例題	**Elementary Number Theory - Power**	(AOJ)

2 つの整数 m, n について、m^n を 1 000 000 007 で割った余りを求めよ
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=NTL_1_B&lang=jp

5.A.1　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．

C++

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

```
3 };
```

表示も作っておこう

C++
```
1 void show(Matrix2x2 A) {
2     cout << "[␣" << endl
3          << A.a << '␣' << A.b << endl
4          << A.c << '␣' << A.d << endl
5          << "]" << endl;
6 }
```

■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．

C++
```
1  // returns C = A*B
2  Matrix2x2 mult(Matrix2x2 A, Matrix2x2 B) {
3      Matrix2x2 C = {0}; // 0 で初期化
4      C.a = A.a * B.a + A.b * B.c;
5      C.b = A.a * B.b + A.b * B.d;
6      C.c = A.c * B.a + A.d * B.c;
7      C.d = A.c * B.b + A.d * B.d;
8      return C;
9  }
10 // (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:

C++
```
1  Matrix2x2 A = {0,1, 2,3}, B = {0, 1, 2, 0};
2  show(A);
3  show(B);
4  Matrix2x2 C = mult(A, B);
5  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．

C++
```
1  // O = A^p
2  Matrix2x2 expt(Matrix2x2 A, int p) {
3      if (p == 1) {
4          return A;
5      } else if (p % 2) {
6          Matrix2x2 T = expt(A, p-1);
7          return mult(A, T);
8      } else {
9          Matrix2x2 T = expt(A, p/2);
10         return mult(T, T);
11     }
12 }
```

使用例:

C++

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

5.A.2 練習問題

例題 **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

■行列表現

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

$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ 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．

問題 **One-Dimensional Cellular Automaton** (アジア地区予選 2012)

与えられた規則で遷移するオブジェクトの T 時間後の状態を求めよ．(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:

C++

```
#include <valarray>
const int N=50;
struct Matrix {
    valarray<int> a;
    Matrix() : a(N*N) { a=0; }
};
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;
}
```

第 6 章

動的計画法 (2)

2018-10-31 Wed

先週の補足: 二次元配列の表示

debug などで二次元配列を表示するときには，ループの順序に注意する

C++

```
1  int main() {
2    int A[3][3]; // order:  A[x][y] という添字で使っているとする
3    A[0][0] = 8; A[1][0] = 3; A[2][0] = 4;
4    A[0][1] = 1; A[1][1] = 5; A[2][1] = 9;
5    A[0][2] = 6; A[1][2] = 7; A[2][2] = 2;
6    // 正しい順番 y => x
7    for (int y=0; y<3; ++y) {
8      for (int x=0; x<3; ++x) {
9        cout << A[x][y];
10     }
11     cout << endl;
12   }
13   /*
14   834
15   159
16   672
17   */
18 }
```

順序を誤ると，斜めに対称になったものが表示される

C++

```
1    // 逆転 x => y
2    for (int x=0; x<3; ++x) {
3      for (int y=0; y<3; ++y) {
4        cout << A[x][y];
5      }
6      cout << endl;
7    }
8    /*
```

```
 9    816
10    357
11    492
12    */
```

6.1 二次元の表を用いる動的計画法

部分構造最適性について (口頭).
今週の問題も，漸化式が多少複雑になるが，先週の問題の応用である．

6.1.1 ナップサック問題

価値と重さを持つ宝物がいくつかあるので，ナップサックの制限 (運べる重さの総合) を超えないように価値が高いものを選びたい．なお，液体のように自由に量を調整できる場合は，重さあたりの価値が最も高いものから貪欲に選べば良い．

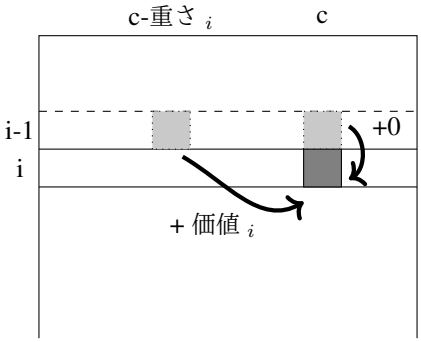
問題	**Combinatorial - 0-1 Knapsack Problem**	(AOJ)

各品物を最大 1 つまで選択できる 0-1ナップサック問題で，制約を満たす価値の合計の最大値を求めよ．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_B&lang=jp

容量が整数で比較的小規模 (配列に確保可能) な場合は，以下の方法が有力である．まず品物に通し番号をつける (並び順は何でも良い)．続いて，部分問題として，品物 i までしかない世界で，ナップサックの重さ制限が c だった場合の，運べる価値の合計の最大値 $V_{i,c}$ を考える．

$$V_{i,c} = \begin{cases} 0 & i \leq 0 \text{ または } c \leq 0 \\ V_{i-1,c} & c - \text{重さ}_i < 0 \\ \max(\text{価値}_i + V_{i-1,c-\text{重さ}_i}, V_{i-1,c}) & \end{cases}$$

$V_{i,c}$ は，i 番目の品物を選ぶ場合と選ばない場合を考慮した最大値であるが，どちらの場合も $V_{i-1,*}$ を用いて計算可能である．そこで，二次元配列を利用して，全体の計算を効率よく行うことが出来る．



問題	**Combinatorial - Knapsack Problem**	(AOJ)

各品物をいくつでも選択できるナップザック問題で，制約を満たす価値の合計の最大値を求めよ．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_C&lang=jp

ほぼ同様の考え方になるが，漸化式が少し変化する．この個数制限のないナップサック問題は，1 個限定のナップサック問題と同じ計算量のオーダ $O(NW)$ で解ける．もし，重さに関する `for` ループを一つ増やして三重ループにする方針をたてた場合は，$O(NW^2)$ になる．改善を考えてみよう．

類題として，各 item 毎に個数制約 (bound) が与えられている，個数制約付きナップサック問題も存在する．スライド最小値の考え方を応用すると，(品物を増やした 0-1 knapsack として解くよりも) 効率的に解を得られる (参考書 [3, p. 302] も参照).

問題	**Coin Changing Problem**	(AOJ)

ぴったり支払うときの，コインの最小枚数を求めよ．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_A&lang=jp

注: 日本のコインは，大きなコインの額を小さいコインが割り切るという性質があるため，大きなコインを使える限り使うほうが良い．つまり，支払額が 500 円を超えていれば，500 円玉を使うのが良い．一方，この問題はそうでない状況も取り扱う．150 円玉，100 円玉，1 円玉という硬貨のシステムで 200 円払うときには，150 円玉を使うと枚数が最小ではない．

6.1.2 列を管理する動的計画法

最大長方形

問題	**Largest Rectangle in a Histogram**	(Ulm Local 2003)

ヒストグラム中に含まれる最大の長方形の面積を求めよ．(long long 注意)
http://poj.org/problem?id=2559
(スタックの回の再掲)

スタックをうまく活用すると，ヒストグラムの本数に比例する手間で求められる．
参考: http://algorithms.blog55.fc2.com/blog-entry-132.html

最長増加部分列

問題	**Combinatorial - Longest Increasing Subsequence**	(AOJ)

最長増加部分列 (Longest Increasing Subsequence) を求めよ.
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_D&lang=jp

最長増加部分列は動的計画法の有名問題の一つで，vector と二分探索 (6.A 節参照) を組み合わせることで $O(N \log N)$ で解くことが出来る．問題末尾の解説参照.

6.1.3 その他

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と良い.

問題	**My friends are small**	(夏合宿 2010)

N 個の荷物がある．重さの合計が W 以下となる荷物の部分集合で，極大なものの選び方は何通り? $N \leq 200, W \leq 10\,000$ http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2333&lang=jp
https://onlinejudge.u-aizu.ac.jp/#/problems/2333

極大とは，選ばなかったどの荷物を加えても W を越える状態を意味する.
考え方: 選ばなかった荷物の

解 説: https://drive.google.com/file/d/1WC7Y2Ni-8elttUgorfbix9tO1fvYN3g3/view

問題	**Barricades**⋆	(Algorithmic Engagements 2007)

問題: 道路の一部をバリケードで塞いで連結な k 都市を他の都市から侵入不能にしたい。
https://szkopul.edu.pl/problemset/problem/1sX3vqLjiqkpxBNI-sh8UC2m/site/

分割統治 + 動的計画法．解法は $O(N^2)$ となるが，証明は少し難しい．$O(N^3)$ の解を丁寧に効率よく実装してみると，良いかもしれない．

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)
補足: $a_0 \cdot n$ は上限まで大きくならないと想定して良い．
https://szkopul.edu.pl/problemset/problem/4CirgBfxbj9tIAS2C7DWCCd7/site/

6.2 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する．
コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である．

C++

```
1 // minmaxsum.cc
2 #if 0
3   所要時間は..であった. 方針は....
4   ...
5 #endif
6 int main() {
7 }
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- 問題 2 題以上を解いて提出する．
  それぞれに，感想 (学んだことや課題にかけた時間など)を添えること
- 「⋆」を含む問題 1 題以上を解いて提出する．
  回答方針や感想 (学んだことや課題にかけた時間など)を添えること

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

6.A 二分探索 binary search

問題を半分に分割して，片方のみを扱えば十分な場合として二分探索をみてみよう．例として，辞書や名簿のように順番に並んでいるデータ列にある要素が存在するかどうかを判定する場合を考える．たとえば “Moore” という姓を，全体が 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 ページと一ページづつ最終ページまで名簿を探すより，速い．名簿のページ数を n とすると，調べるページ数は $O(n)$ と $O(log(n))$ の差がある．

6.A.1 データ列中の値の検索

C++ の標準ライブラリに収められている `binary_search` は整列済の配列から二分探索により要素の有無を判定する．

C++

```
1  #include <iostream>
2  #include <algorithm>
3
4  int S[/*大きな数*/];
5  int main() {
6    ... // S の初期化
7    sort(S, S+L); // 配列 S 内を昇順に並び替えておく
8    ...
9      if (binary_search(S, S+L, a)) {
10       // a が S 内にある
11     }
12 }
```

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`, 実用的には `n==10` 程度に取る場合もある．)
3. 区間の中央を求める `m=(l+r)/2`
4. `value < A[m]` なら (次は前半 `[l,m)` を探したいので)`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 `[m,r)` を探したいので `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．

C++

```
1 bool bsearch(const int array[], int left, int right, int value) {
2     while (left + 1 < right) {
3         int med = (left+right)/2;
4         if (array[med] > value) right = med;
5         else left = med;
6     }
7     return left < right && array[left] == value;
8 }
```

自分で二分探索を実装したら，Search II で動作を確認することができる．

例題　**Search II**　(AOJ)

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 の要素数は 10 万以内，T は 5 万以内．配列に含まれる要素は，0 以上 10^7 以下．

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

問題の仕様に注意

(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．

`S=[1,1,1], T=1` とすると答えは 1.

6.A.2　制約を満たす最小値・最大値を求める

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．

問題　**Search - Allocation**　(AOJ)

様々な重さの荷物を，並べられた順にトラックに積むことを考える．荷物の重さは並べられた順に配列 `w` に格納してあるとする．全てのトラックの最大積載量は同じとする．`k` 台のトラックにすべての荷物を順に全て積むためには，各トラックの最大積載量は最低どれだけ必要かを求めたい．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_4_D

ヒント: トラックの最大積載量 P として二分探索し，全ての荷物を積める最小値を求める．積みきれる最小値がほしいので，「積みきれない最大値」が存在する範囲を `[l,h)` で表現し，上限 `h` を減らす条件を `ok` として表現する．求める答えは `h` となる．

C++

```
1  bool ok(int P) {
2    ... // 最大積載量 P のトラックに前から順に積み込んだ時の必要台数が K 以内かを返す
3  }
4  int main() {
5      // 問題読み込み
6      int l = 0, h = // 大きな数;
7      while (l+1 < h) {
8          int m = (l+h)/2;
9          if (ok(m)) h = m; else l = m;
10     }
11     printf("%d\n", h);
12 }
```

関数 ok は，たとえば以下のように作成できる:

- 現在のトラックに積んだ重さ (初期値 0) とトラック台数 (初期値 1) を表す変数を作成
- 各荷物について，現在のトラックに積んで最大積載量を超えないならそのまま積む (現在のトラックに積んだ重さを増やす)/そうでなければ新しいトラックに積む (トラックの台数を増やして，現在のトラックに積んだ重さをその荷物に設定)
- 最後に台数を K と比較

第 7 章

最小重み全域木

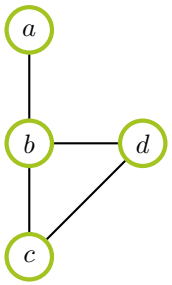
2018-11-07 Wed

概要

本日から複数回，グラフの話題を扱う．今週は，グループを管理する Disjoint Set (Union-Find Tree) というデータ構造を紹介する. またそれを使ってグラフの最小重み全域木を求めてみよう.

7.1 グラフと木

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.



定義: 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．辺の向きを区別するグラフを有向グラフ，区別しない場合を無向グラフと呼ぶ.

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 に対して，v の接続辺の個数 $|E(v)|$ を次数 $d(v)$ という.

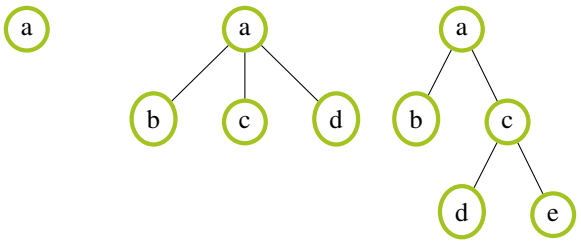
例: 三角形の各頂点の次数は 2

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．(新たに定義を導入するまでは) 隣接する頂点をリストにして並べたものをパス (path) と呼ぶことにする.[*1]

[*1] 辺や頂点の重複を許すかどうかなど, trail, walk など異なる用語を用いるので，詳しく学ぶ際には注意

7.1.1 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．



木 1　　木 2　　木 3

定義

無向グラフ G に対して，全ての 2 頂点 v, w に対して $v - w$ 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

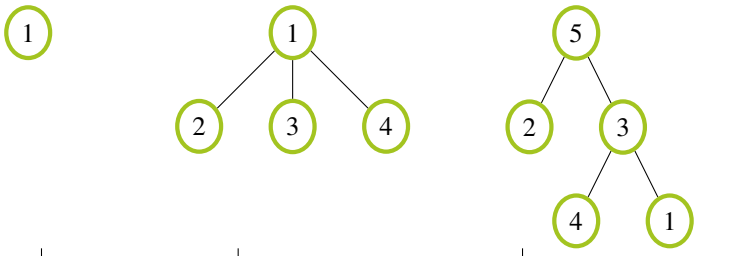
木の頂点の特別な一つを根 (root) と呼ぶ．グラフを図示する際には，根を一番上または下に配置することが多い．次数 1 の点を葉 (leaf) と呼ぶ．木の辺で結ばれた 2 頂点について，根に近い方の頂点 (あるいは根) を親 (parent) と呼び，遠い方を子 (child) と呼ぶ．

7.2 計算機での木の表現

N 個の節点を持つ「木」を計算機上で表す方法の一つは，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) で管理する方法である．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親の番号を，親を持たない場合 (すなわち根 (root)) には自分自身または -1 などの特殊な番号を割り当てる．

なお，各節点に対応する子は複数可能性があるので子を管理する場合は，通常のグラフと同様に，一次元配列より複雑な表現 (隣接リスト，隣接行列など) が必要となる．

木



配列表現

i	1
P[i]	1

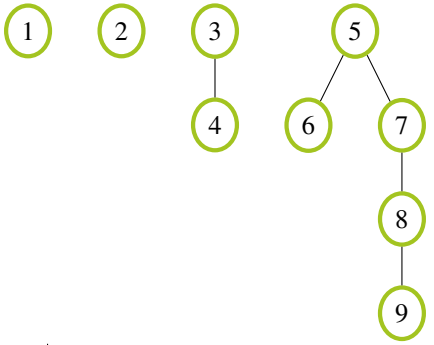
i	1	2	3	4
P[i]	1	1	1	1

i	1	2	3	4	5
P[i]	3	5	5	3	5

7.3 Disjoint Set (Union-Find Tree)

二つの要素が同一グループに属するかどうかを効率的に判定する手法の一つが，Disjoint-set (参考書 [3]p.81-) である．Union Find Tree とも呼ばれる．

■データ表現



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

木の集合を，配列を使って表現する．

■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．この判定を，要素が属する木の根 (root) を比較することで行う．

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. 2 と 4 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，出力は変わらない．(実際には小

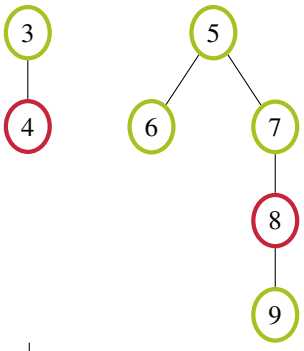
さな (低い) 木を大きな (高い) 木の下につける方が効率が良い，が，この資料では無視する)

i	1	2
P[i]	1	2

元のグラフ

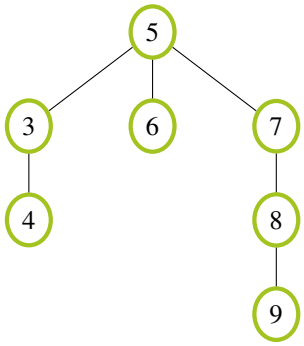
i	1	2
P[i]	1	*1*

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

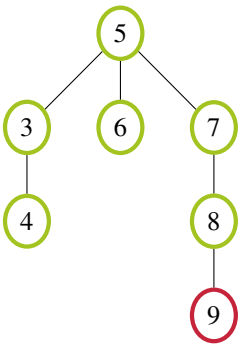


i	3	4	5	6	7	8	9
P[i]	*5*	3	5	5	5	7	8

4 と 8 を併合したグラフ

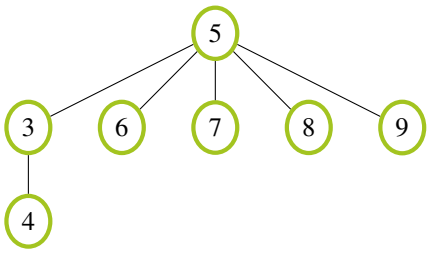
(グループ全体を併合するためには根同士を接続)

■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．)

C++

```
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;
    // 注： 通常は [0,N-1] または [1,N] の範囲を使うが
    // どちらでも対応できるように資料では [0,N] の範囲とした
}
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:

C++

```
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;
}
```

例題 **Disjoint Set: Union Find Tree** (AOJ)

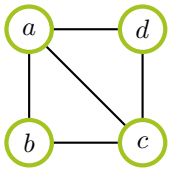
Disjoint Set でグループを管理せよ

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DSL_1_A&lang=jp

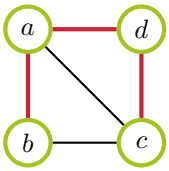
7.4 全域木

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．

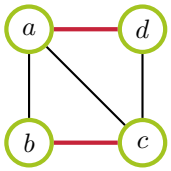
メモ: グラフが連結なら全域木が存在する．全域木は一つとは限らない (完全グラフの場合は n^{n-2} 個もある)．最小全域木も一つとは限らない．最小全域木を求める問題は，最小全域木のうちの一つを求める問題を指すことが多い．



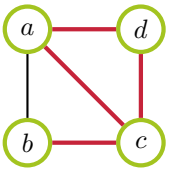
元のグラフ



赤い部分グラフは全域木



全域木でない (非連結)



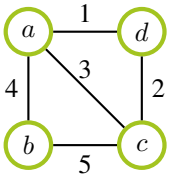
全域木でない (閉路)

7.4.1 クラスカル法

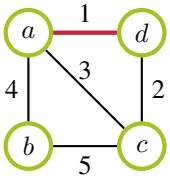
最小重み全域木を求める方法として，有名な方法にクラスカル法とプリム法があるが，ここでは前者を紹介する．

クラスカル法は以下のように動作する (参考書 [3] p.101-):

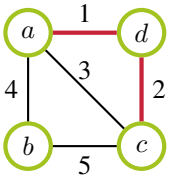
- 辺を重みの小さい順にソートする
- T を作りかけの森 (または林, forest; 最初は空，閉路を含まないグラフ，連結とは限らない) とする
- 重みの小さい順に，各辺 e に対して以下の操作を行う
  $T + e$ (グラフ T に辺 e を加えたグラフ) が閉路を含まなければ T に e を加える



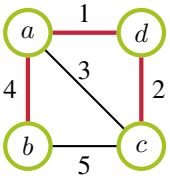
元のグラフ



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．

例題 **Graph II - Minimum Spanning Tree** (AOJ)

与えられたグラフの最小全域木の重みの総和を求めよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_12_A

考え方:

- グラフが隣接行列だけ与えられるので，三角行列 a_{rc} $(r < c)$ 部分だけ考えて，値が`-1` でないものに通し番号をつける．そのような `i` 番目の辺について，`cost[i]`, `src[i]`, `dst[i]` を記録する。
- 辺のコストと番号のペア `pair<int,int>` を配列や `vector` に格納する
- 上記で作成した列を `sort` する

C++

```
1  int M = 0; // 見つけた辺の数
2  for (int r=0; r<...)
3    for (int c=c+1; c<...)
4      if (A[r][c] > -1) {
```

```
5           cost[M] = A[r][c];
6           src[M] = r
7           dst[M] = c
8           M += 1
9         }
10    pair<int,int> edges[...];
11    // edges にデータを入れて，コストの小さい順に整列する
```

■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．

C++

```
1           int 合計 = 0;
2           for (int i=0; i<M; ++i) { //安い辺から順に
3               if // (辺の両端の節点が既に連結だったら)
4                   continue;
5               // 辺の両端の節点を同一グループに併合する;
6               // 合計に，辺のコストを加える;
7           }
8           // 合計を出力;
```

7.5 練習問題

問題	**True Liars**	(アジア地区予選 2002)

真実のみいう人 (=正直者) と嘘だけ言う人住む島があって，それぞれの合計人数はわかっている．「人 x が人 y を正直者である/ない」と言ったという情報を総合して，割り当てが一意に定まるならばそれを求めよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1238

参考: a が b を正直者と評した場合，(a,b) は両者正直者もしくは両者嘘つきである．
ヒント (白い文字で):

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重み付き Union-find 木を作れば良い．

問題	**Slim Span**	(アジア大会 2007)

最小の辺の重みと最大の辺の重みの差が最小の全域木を求めよ.
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1280

問題	**Sinking islands**	(模擬国内予選 2013)

沈みゆく島にどのように (問題文参照) 橋をかけるかを求める.
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2511

問題	**There is No Alternative**	(アジア地区予選 2014)

どのような最小重み全域木を作っても必ず使う辺を求める.
http://judge.u-aizu.ac.jp/onlinejudge/cdescription.jsp?cid=ICPCOOC2014&pid=F
c.f. 類題 http://codeforces.com/problemset/problem/160/D

問題	**Byteland**	(1st Junior Polish Olympiad in Informatics)

問題文参照
https://szkopul.edu.pl/problemset/problem/JXaPu39wQfiZaqlM51zlchez/site/

7.6 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する.

コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である.

C++

```
1 // minmaxsum.cc
2 #if 0
3   所要時間は..であった. 方針は....
4   ...
5 #endif
6 int main() {
7 }
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- 例題 “Disjoint Set: Union Find Tree” と “Minimum Spanning Tree” の両方を解いて提出する．
  それぞれに，感想 (学んだことや課題にかけた時間など)を添えること
- 問題 (または例題) から 2 題以上を解いて提出する．
  回答方針や感想 (学んだことや課題にかけた時間など)を添えること

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

第 8 章

幅優先探索

2018-11-21 Wed

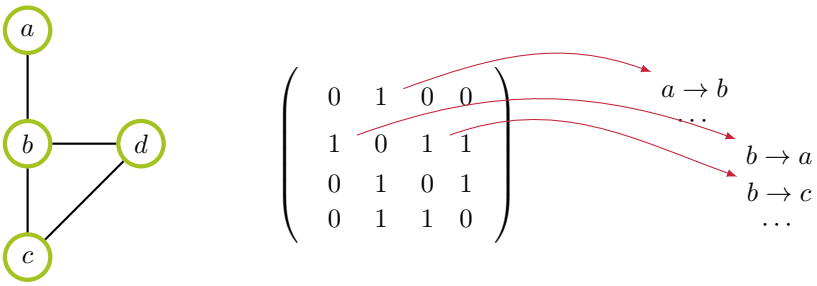
概要

まずは，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) を紹介する．
今週の内容に取り組む前に，キューの使い方 (4.2.2 章)，グラフと木の概念 (7.1 章) を思い出しておくこと．

8.1 グラフの表現: 隣接リストと隣接行列

各節点の親を覚えておくと木構造で親をたどることができたが，一般のグラフの節点を訪問するためには，より便利なデータが必要となる．

■隣接行列　初めに，グラフを隣接行列(adjacency matrix) で表現する方法を紹介する．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．

グラフの表現の中で，隣接行列は比較的「贅沢な」グラフの表現方法である．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．

■隣接リスト　隣接リストは，接続先の頂点リストを図のようなリストで表す．実際には，頂点名 a,b,c,d は頂点番号 0,1,2,3 を用いると管理が容易である．C++ の場合は頂点数を N として `vector<int> edges[N];`

として，edges[i] が i 番目の頂点の接続先を表すようにする．先の章で辺のコストを管理する場合は，行き先とコストをペアにして，vector<pair<int,int>> edges[N]; などとする．

$$
\begin{aligned}
a &\rightarrow b \\
b &\rightarrow a \ \ c \ \ d \\
c &\rightarrow b \ \ d \\
d &\rightarrow b \ \ c
\end{aligned}
$$

隣接リストは，辺の数に比例したメモリを使用する．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，頂点に比べて辺が疎であることが多い．隣接リストは，そのようなグラフに適する．(参考: 参考書 [3, pp. 90, 91])

例題 **Graph** (AOJ)

有向グラフの隣接リストが与えられるので，隣接行列に変換する．

初めにグラフの節点の数 N が与えられ，それに続いて N 行が与えられる．各行は一つの節点に対応し，その節点から出ている辺を表す．初めの数 u が節点の番号，続く数 k が辺の数，その後に続く k 個の数が各辺の行き先に対応している．(各行に含まれる数字の個数は，行毎に異なりうる)

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_11_A

この問題のように多くのケースでは，頂点番号を $1, \ldots, N$ でつけている．一方，C++ などでは配列の添字は 0 から始まるので，(1) 配列を多めに確保して添字 0 に対応する部分を無視する，(2) 添字番号全てから 1 を引いて処理する，などで対応する．

Python による回答例を示す．この資料では，問題文中に出てくる変数を (あとで値を変更する必要がない場合は) 大文字で表記し定数として扱う．

Python3

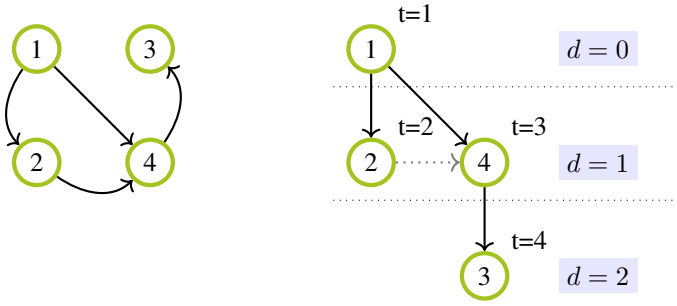
```
1 N = int(input())
2 G = [[0 for _ in range(N)] for _ in range(N)] # NxN の 2 次元配列
3 for _ in range(N):
4     u,k,*varray = map(int,input().split()) # u,k は数, varray は配列
5     for v in varray:
6         # G において u-1 と v-1 をつなげる
7 for row in G:
8     print('␣'.join(map(str,row))) # G の各行を出力
```

8.2 幅優先探索 (BFS)

問題	**Breadth First Search**	(AOJ)

節点 1 を始点に幅優先探索を行い，各節点の始点からの距離を表示せよ
(入力の形式は例題 “Graph” と同じ)
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_11_C

幅優先探索(参考書 [3]p. 36) は，出発地に近い頂点から順に訪問する．



入力のグラフ (サンプル入力)　　幅優先探索の作る木

図の左に示したグラフが入力として与えられたとして，頂点 1 を始点に幅優先探索を行った例を右に示す．右図の木の各節点の添字 t は訪問時刻，実線は幅優先探索中に使われた辺，点線は無視された辺を表す．d は始点からの距離である．幅優先探索において，距離が等しい頂点の訪問順序は任意である．

このような探索を実現するために，キューというデータ構造 (第 4 回参照) 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．

1. 始点から各頂点 n への距離 $d_n = \infty$ と設定
2. キューに始点を入れる．(始点から) 始点への距離を設定: $d_{\text{始点}} = 0$
3. キューが空でない間，以下を繰り返す
   (a) キューの先頭の頂点 n を取り出す
   (b) n から移動可能な各頂点 n' について，$d_{n'} = \infty$ であれば (初めて見つけた頂点なので) 以下を行う
      i. n' をキューに入れる
      ii. n' への距離を設定: $d_{n'} = d_n + 1$

この問題の入力の形式は例題 “Graph” と同じで，サンプル入力も全く同じなので，入力は作成済で，`G[s][t]==1` の時に (かつその時に限り)，節点 `s` から節点 `t` に移動可能とする．

Python3

```
1 import collections
2 D = [-1 for _ in range(N)]
3 D[0] = 0 # 始点への距離は 0, 他の距離は -1
4 Q = collections.deque()
```

```
5  Q.append(0)                          # 始点
6  while len(Q) > 0:
7      print("bfs", Q)  # 各ステップでの Q の動作を確認 (後で消すこと)
8      cur = Q.popleft()
9      for dst in range(N):
10         if ...: # cur から dst に移動可能かつ、dst が未訪問だったら
11             D[dst] = D[cur]+1
12             Q.append(dst) # Q に dst を詰める
13 for v in range(N):
14     print(v+1, D[v])          # [0,N-1] から [1,N] に変換
```

実行例は以下のようになる

```
bfs deque([0])
bfs deque([1, 3])
bfs deque([3])
bfs deque([2])
1 0
2 1
3 2
4 1
```

キューには始点から辿れる節点が順に，各一度だけ，入れられる．8 行目の条件で，dst が未訪問かどうかを判別しないと，合流やループがあるグラフで問題が生じる，(たとえば 1 から 2 に辺があり，2 から 1 にも辺がある場合に，1-2-1-2-1 と移動し続け無限ループしてしまう)．未訪問かどうかは，D[dst] を見ると判別できる．

C++

```
1  #include <queue>
2  int D[...];
3  void bfs(int src) {
4      cerr << "bfs root = " << src << endl;
5      queue<int> Q; // 整数を管理するキューの定義
6      Q.push(src);
7      D[src] = 0; // 出発点
8      while (! Q.empty()) {
9          int cur = Q.front(); // 先頭要素を取り出す
10         Q.pop();
11         // 動作確認用表示
12         cerr << "visiting " << cur << ' ' << D[cur] << endl;
13         for (...) { // 各行き先 dst に対して
14             if (..) { // cur から dst に辺があり，dst が未訪問なら
15                 D[dst] = D[cur]+1; //
16                 Q.push(dst); // dst を訪問先に加える
17             }
18         }
19     }
20 }
```

8.3 幅優先探索の応用

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) を行うことで解ける問題もある．

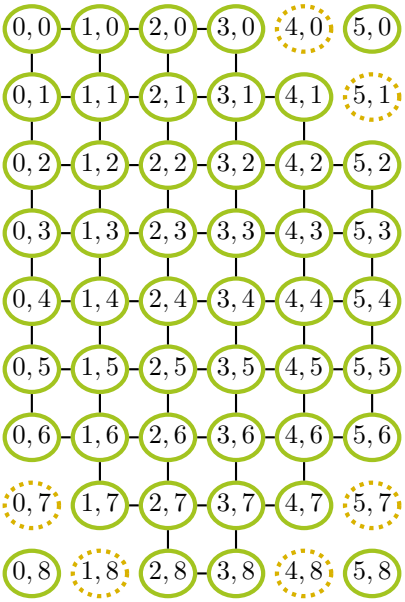
- 連結性: ある頂点から BFS を行いすべての頂点を訪問できれば連結．そうでなければ，複数の非連結のグラフからなる．(無向グラフの場合)
- 二部グラフかどうかの判定: 辺の両端の色が異なるように赤と青で塗り分けられるか．BFS で色を塗りながら辺をたどり矛盾がないかを調べれば良い

8.3.1 連結性

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．

C++

```
1 bool valid(int x, int y) {
2    return xが[0,W]の範囲
3         && yが[0,H]の範囲
4         && (x,y)が壁でない;
5 }
```

■回答骨子: '@' の位置 (x,y) を探してそこから，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．

問題	島はいくつある？	(国内予選)

島の数を数える
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1160&lang=jp

ほぼ同じ

8.3.2 二部グラフの判別

二部グラフは各辺の両端の頂点の色が異なるように，全体を二色で塗り分けられるようなグラフである (職種と応募者，安定結婚問題などで登場する)．BFS で色を塗りながら辺をたどり矛盾がないかを調べれば良い．

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

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．

8.3.3 応用問題

問題	**Weather Forecast⋆**	(アジア地区 2003)

2x2 の雲を操って，適度に雨を降らせる方法を探せ．各土地に 7 日に 1 度は雨を降らすこと，また祭りの日に雨を振らせては行けない．雲の動きなどは原文参照．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1243

ヒント:

問題	**Karakuri Doll⋆**	(模擬国内予選 2007)

からくり人形師の JAG 氏作の人形の動作を確認せよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2017

8.4 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する．

コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である．

C++

```
1 // minmaxsum.cc
2 #if 0
3   所要時間は..であった．方針は....
4   ...
5 #endif
6 int main() {
7 }
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- “Red and Black” を含む 2 題 (たとえば “Breadth First Search”) を解いて提出する．
  それぞれに，感想 (学んだことや課題にかけた時間など)を添えること
- “⋆” つきの問題を 1 題以上解いて提出する．
  回答方針や感想 (学んだことや課題にかけた時間など)を添えること

提出先: ITC-LMS
提出の注意:

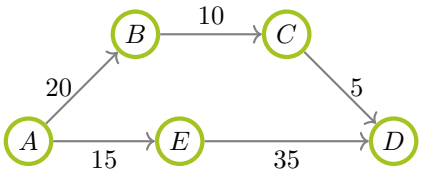
- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

第 9 章

最短路問題

2018-11-28 Wed

概要

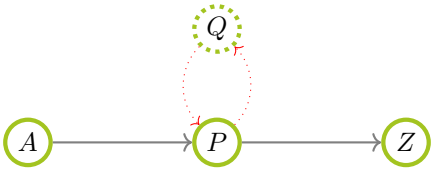


A から D まで最も安くて幾らで行ける? A..E は町．町をつなぐ道路は規定の料金がかかる．➔ 経路 {A,B,C,D} がコスト最小で 35 (最初に E に移動すると損)

9.1　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．

コストは非負であると仮定する (通行料を払うことはあっても報奨金をもらうことはない)．すると，最短路としては同じ頂点を二度以上通るものは考慮から外して良い．たとえば下図のように $A,\ldots,P,\ldots,Q,\ldots,P,\ldots,Z$ と P を二回通るルートについて考えると，途中の $P,\ldots,Q,\ldots,P$ と回るコストは非負であるので，それを除いた $A,\ldots,P,\ldots,Z$ というパスだけを考えれば良い．



一般の場合の議論はアルゴリズムの教科書を参照されたい．

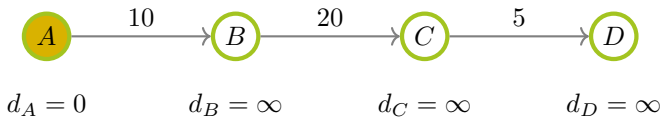
大きなグラフも扱うので，グラフの表現としては，隣接リストが適している (前回参照)．C++ で接続関係を

隣接リストで表現するには，各頂点から接続する頂点を標準ライブラリの vector<int>で表すことが多い．すなわち，N 個の頂点があるとして vector<int> E[N]; というデータを定義し，頂点 i から接続する頂点を E[i] で表す．さらに，今回のテーマでは，辺はコストを持つので，(1) ⟨ 移動先，コスト ⟩ を組みにして，vector<pair<int,int>>で表現する，もしくは (2) 辺 id を vector<int>で管理して，辺 id に対応する，始点，終点，コストを配列 int SRC[L], DST[L], COST[L]; などで別に管理する (L は辺 id の最大値)，などの方法を取る．

9.2 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．たとえば，往復のコストを求める場合は，往路と復路のそれぞれで単一始点最短路問題を 2 回解くと解が得られる．

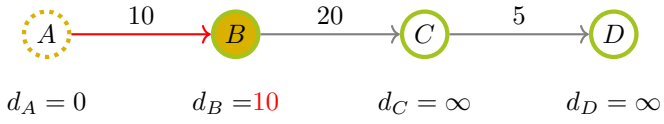
9.2.1 緩和 (relaxation)



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．

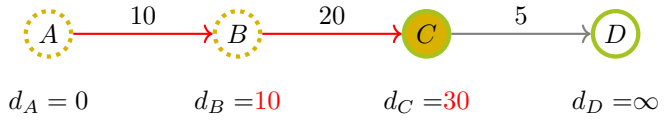
定義: $d[x]$ を A から x までの最短コストの上限とする．

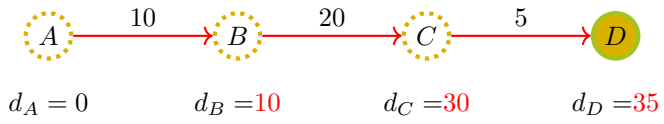
初めに，$d[A] = 0$ (A から A まではコストがかからない)，$d[B] = d[C] = d[D] = \infty$ (情報がないので ∞) と定める．



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．

9.2.2 Bellman-Ford 法

(参考書 [3, 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 ∞
  $d[v_s] \leftarrow 0$ ▷ 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

- 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)

9.2.3 Dijkstra 法

(参考書 [3, 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 ∞
  $d[v_s] \leftarrow 0$ ▷ 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
    select u $s.t.$ $\arg\min_{u \in Q} d[u]$ ▷ 「最短距離が未確定の頂点」で $d[u]$ が最小の u を選択
    $S \leftarrow S \cup \{u\}, Q \leftarrow Q \setminus \{u\}$ ▷ u までの最小距離は確定
    for $v \in Q$ $s.t.$ $(u,v) \in E$ **do**
      $d[v] \leftarrow \min(d[v], d[u]+w(u,v))$ ▷ 緩和

- 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)

9.2.4 例題

問題	**Single Source Shortest Path I**	(AOJ)

都市 0 から全ての都市への最短路を求めよ．都市には 0 から $|V|-1$ までの番号がついている．入力は，1 行目に都市の数 $|V|$，続く $|V|$ 行に各節点からの接続情報が与えられる．

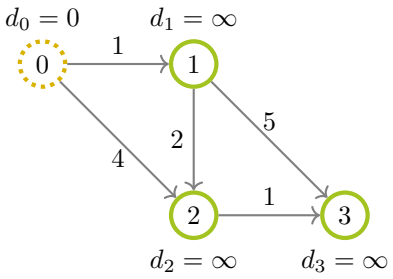
詳しくは問題文を参照．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_1_A&lang=jp

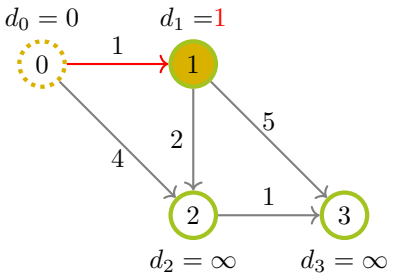
グラフの表現

この問題では都市の数が最大 100 000 と多い．これを隣接行列で表そうとすると，$(100\,000)^2$ の要素数が必要となり，メモリ制限を確実に越える (試算せよ)．一方，辺の数は最大 500 000 と $(100\,000)^2$ より遥かに小さいので，辺毎に管理すると良い．Bellman-Ford 法の場合は，問題文の指定通りに始点と終点を管理すれば十分である． Dijkstra 法を用いる場合は，`vector` を用いて隣接リスト形式にすると都合が良い．

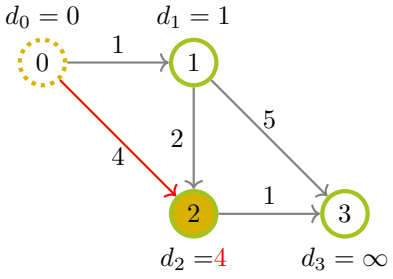
■Bellman-Ford 法で解いてみよう　サンプル入力 1 のグラフに対して，与えられた辺の順に処理を行った場合の動作は次のようになる．(この例では偶然全ての辺を一度づつ見るだけで最短距離が求まったが，グラフの形と辺の並び順に応じて一般には $|V|-1$ 回必要である．)



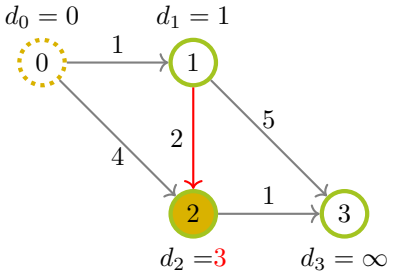
サンプル入力 1 の初期状態



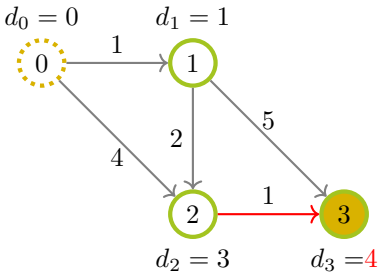
辺 {0,1} で緩和: $d_0 + 1 = 1 < \infty$



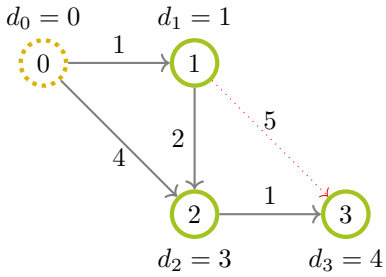
辺 {0,2} で緩和:$d_0 + 4 = 4 < \infty$



辺 {1,2} で緩和: $d_1 + 2 = 3 < 4$



辺 {2,3} で緩和: $d_2 + 1 = 4 < \infty$



辺 {1,3} では更新が起こらない: $d_1 + 5 = 6 > 4$

C++

```
int V, E, R, S[500010], T[500010], D[500010]; // 問題で与えられる入力
int C[100010]; // 各頂点までの最短距離の上限
// 無限を表す定数を全頂点をたどる最大超に設定
const int Inf = 10000*100000+100;

int main() {
  cin >> V >> E >> R;
  for (int i=0; i<E; ++i)
    cin >> S[i] >> T[i] >> D[i]; // 各辺を入力
  ... // C を初期化： C[R] を 0 に，他を Inf に
  for (int t=0; t<V; ++t) { // V 回繰り返す
    bool update = false;
    for (int i=0; i<E; ++i) {
      int s = S[i], t = T[i], d = D[i]; // i 番目の辺の s,t,d に対して
      if (C[s] < Inf && ...) { // 辺 s,t で緩和できるなら
        C[t] = ...// C[t] を更新;
        update = true; // 更新が起こったことを記録
      }
    }
    if (!update) break; // 辺を一巡して更新がなければ打ち切って良い
  }
  ... // 出力
}
```

Python3

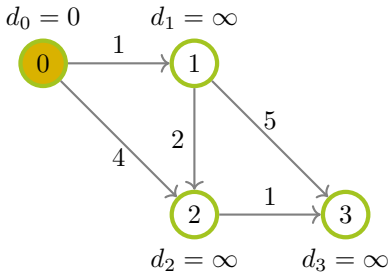
```
NV,NE,R = map(int,input().split()) # NX で X の個数を表した
Inf = 1001001001 # 全頂点を通るパスより大きな値
E = [tuple(map(int,input().split())) for _ in range(NE)]
# 初期化
D = [Inf for _ in range(NV)]
D[R] = 0
# メインのループ
for t in range(NV):
    update = 0
    for s,t,d in E:
        # try to decrease D[t] w.r.t. edge (s,t) with cost d, here
        # increment update if D[t] was changed (decreased)
    if update == 0:
```

```
14            break
15    for v in range(NV):
16        print(D[v] if D[v] != Inf else "INF")
```

■Dijkstra 法で解いてみよう　Dijkstra 法 (9.2.3 節) で解く場合は，手順 8 行目で，Q 内で距離のもっとも近い都市 u を取り出す $\arg\min_{u\in Q} d[u]$ の部分の処理が重要である．単純には，V の頂点全てを for 文で調べて，条件に該当するかどうかを判別すれば良い．この実装では $O(V^2)$ の計算量となる．

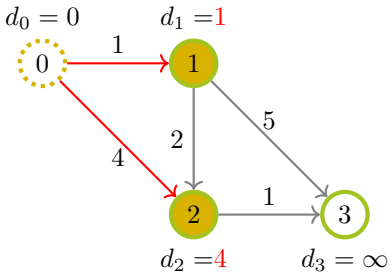
辺が疎な場合は，優先度つきキュー (**priority queue**, 以前の資料参照) を使うと効率が良くなる ($O(E \log V)$) 場合がある．C++ には標準ライブラリとして priority_queue があるのでそれを使うと良い．始点からの距離と都市のペア pair<int,int>を管理し，手順 8 行目で，距離の近い順に都市を取り出す．そのために，手順 11 行目で更新が行われた頂点を優先度つきキューに push する．本来はキュー内部にある頂点の距離を減らすことが出来ると効率が良いが，多くの標準ライブラリの実装では難しい．代わりに重複して push し，二度目以降に取り出した頂点は無視する．注意点として，C++ の標準ライブラリの **priority_queue は大きい順**に要素を取り出すので，比較関数を指定して動作を変更するか，距離の符号を反転させて与えるような工夫が必要となる．

優先度つきキューで実装した Dijkstra 法の動作例を，以下に示す．優先度つきキュー P は，$\langle$ 始点からの距離, 頂点番号 $\rangle$ のペアを要素として持ち，距離の小さい順に，先頭要素 (左端) が取り出されるとする．図のグラフ 1 つが手順 7 行目からのループの 1 回の実行に対応する．



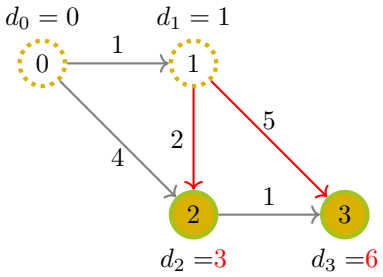
初期状態

$P = (\langle 0, 0\rangle)$



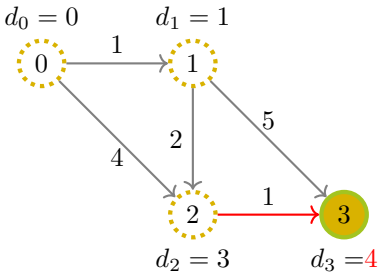
$\langle 0, 0\rangle$ を取り出し頂点 0 を確定，頂点 1, 2 を緩和

$P = (\langle 1, 1\rangle, \langle 4, 2\rangle)$



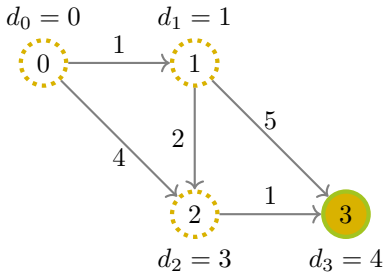
頂点 1 を確定して頂点 2, 3 を緩和

$P = (\langle 3, 2\rangle, \langle 4, 2\rangle, \langle 6, 3\rangle)$



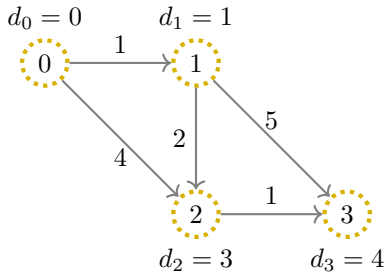
頂点 2 を確定して頂点 3 を緩和

$P = (\langle 4, 2\rangle, \langle 4, 3\rangle, \langle 6, 3\rangle)$



頂点 2 は既に確定しているので $\langle 4, 2\rangle$ は無視

$P = (\langle 4, 3\rangle, \langle 6, 3\rangle)$



$\langle 4, 3\rangle$ を取り出し頂点 3 が距離 4 で確定

$P = (\langle 6, 3\rangle)$

9.3 参考: 全点対間最短路

(この節は決して難しくはない–むしろ易しい–が，演習時間の関係で参考扱いとする)

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，全ての節点間の最短路を求める Floyd-Warshall を覚えてくとよい (参考書 [3, 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
```

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.

9.3.1 例題

例題 **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．

C++
```
1  #include <iostream>
2  #include <cstdio>
3  using namespace std;
4  int N, M, A, B, C, D, x1, x2, y1, y2;
5  int main() {
6      scanf("%d%d", &N, &M);
7      for (int i=0; i<M; ++i) {
8          scanf("%d,%d,%d,%d", &A, &B, &C, &D);
9          cerr << "read␣" << A << '␣' << B << '␣' << C << '␣' << D
10             << endl;
11         // A → B がコスト C
12         // B → A がコスト D
13     }
14 }
```

■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．

C++
```
1  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．(手順中 5, 6 行目で加算を行うため)

C++
```
1  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.

9.3.2 負の重みを持つ辺がある場合 (参考)

負の重みを持つ辺 (負辺) としてモデル化することが適切な場合もありうる．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を考えよう．そのような場合に，最短路の概念はどう変化す

るだろうか? 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，負閉路(negative weight cycle) 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に最短路が定義できない．始点から終点までの途上に負閉路が無ければ最短路は定義可能で，(正ではないかもしれないが) 最小のコストが定まる．

Floyd-Warshall 法は幸い負閉路があっても動作し，終了時各点 i について $K[i][i] < 0$ であるなら点 i を含む閉路が存在する．ただし実装にあたっては，辺の有無に注意を払う必要がある．

9.3.3 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．制限が 1 秒程度であるとすると，$V = 100$ 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．

もし重みが負の辺があると，Dijkstra 法は正しく動作しない． Bellman-Ford 法は最短路が定義される場合には正しい解を発見可能で，負閉路の存在は頂点の個数 $|V|$ 回目の更新を試みて成功するかどうかで確認できる．

9.4 応用問題

問題	**Railway Connection**$\star$	(国内予選 2012)

最も安い運賃の経路を求める

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1182&lang=jp

ヒント: 距離と料金の二つの観点があるので，それぞれ対応する．(つづきは白文字で)

問題	**崖登り** $\star$	(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

問題	壊れたドア ⋆	(国内予選 2011)

どこかのドアが壊れている条件で，最も都合が悪い場所で壊れたドアが見つかった時点からの迂回を考慮した最短路を求める．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1178&lang=jp

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)

https://szkopul.edu.pl/problemset/problem/5g0vDW-MvMGHfWQqh56jQKx1/site/

9.5 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する．

コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である．

C++

```
1 // minmaxsum.cc
2 #if 0
3   所要時間は..であった．方針は....
4   ...
5 #endif
6 int main() {
7 }
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- “Single Source Shortest Path I” を Bellman-Ford 法と Dijkstra 法の両方で解く．(Dijkstra 法がどうしても難しい場合は，“Single Source Shortest Path I” を Bellman-Ford 法で解いたうえで，加えて “Reward for Carpenter” を解くことにしても良い)
- 応用問題から 1 題解く

提出先: ITC-LMS

提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨す

るが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．

- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

第 10 章

深さ優先探索と構文解析

2018-12-05 Wed

概要

深さ優先探索を中心に再帰的技法を学ぶ．深さ優先探索は，少し前に扱った幅優先探索とならぶ，グラフを扱う有力な手法である．後半は，“(3+5)*4” のような式を木に分解する，再帰下降という手法を紹介する．

10.1　再帰的処理

■再帰関数　再帰関数について，以下のように帰納的定義に対応する関数と理解してきた．

(夏学期資料の再掲) 例として，$1+2+\cdots+n$ を求める関数を定義してみよう．この関数を sum(n) と書くことにすると，次のように帰納的に定義できる．

$$\mathrm{sum}(n) = \begin{cases} 1 & n=1 \text{ のとき} \\ n+\mathrm{sum}(n-1) & \text{それ以外} \end{cases}$$

この定義の中では sum 自身を使っていることに注意．これは，ほとんどそのまま C++ の定義に置き換えることができる:

C++

```
1  int sum(int n) {
2    if (n == 1) return 1;
3    else return n + sum(n-1);
4  }
```

赤字の sum が，
再帰の構造になっている．

再帰関数を理解するコツは，再帰で呼びだされている部分 `sum(n-1)` が計算されると，(計算方法の詳細はともかく)「1 から n-1 までの合計に置き換わる」と一度信じてから検証することである．

■再帰的手続き　同様に，再帰的手続きについて考えよう．この資料では，関数を計算の結果値を返すもの，手続きを副作用を行うことが主であるものと呼び分けることにする．手続きで行う副作用の例としては，「表示」やグラフの節点への訪問時刻の「記録」などが挙げられる．

以下の例題を考えてみよう．もし苦労する場合は，ITC-LMS 内の参考資料 (過去の授業教材) summer.pdf の再帰の部分 (第 5 章) をよく復習のこと．

例題　**m 進数**　(金子)

整数 m, n が与えられる ($1 \leq n \leq 8, 1 < m \leq 10$). n 桁の m 進数を小さい順に全て表示するプログラムを作成せよ.

この問題は各自で入出力を作ってテストすること.

```
% cat sample.txt # あらかじめ作っておく
3 2
%  ./a.out < sample.txt
00
01
02
10
11
12
20
21
22
```

簡単のため $m = 2$ すなわち 2 進数を考える. 正の整数 n に対して所要の目的を達する手続きを, n-1 に対して同様の処理を行う手続きを元に組み立てたい. n 桁の 2 進数は左端の 1 文字と残り n-1 桁の 2 進数に分解できる. また, 左端は 0 または 1 である. そこで引数 prefix と n に対して,「n 桁の 2 進数を全て生成して, それぞれに prefix をつけて表示する」という関数を `X(prefix, n)` とする.

$$\texttt{X}(\texttt{prefix}, n) = \begin{cases} \text{prefix を表示} & (n = 0) \\ \texttt{X}(\texttt{prefix} +' 0', n-1) \text{ を実行後に} & \\ \quad \texttt{X}(\texttt{prefix} +' 1', n-1) \text{ を実行} & (n > 1) \end{cases} \tag{10.1}$$

`X("", n)` が, n 桁の 2 進数を全て表示することを確認せよ. 2 進数を拡張して m 進数を扱うには, `for` 文などを使えば良い.

再帰的手続きを理解するコツは, 再帰で呼びだされている部分 `X(prefix, n-1)` の処理が終わると, (方法の詳細はともかく)「担当範囲の表示が終わっている」と, 一度信じてから検証することである.

文法: C++ で, 整数 a を N 桁で表示し, 桁が足りない際に 0 を補うには以下のように行う.

C++

```
#include <iomanip>
void show(int N, int a) {
  cout << setw(N) << setfill('0');
  cout << a << endl;
}
```

10.2 深さ優先探索 (DFS)

問題	**Depth First Search**	(AOJ)

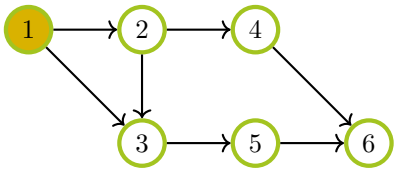
番号の若い順に節点を深さ優先探索で訪問する時に，その訪問順序を表示せよ

「未発見の頂点が残っていれば、その中の 1 つを新たな始点として探索を続けます。」ことに注意.

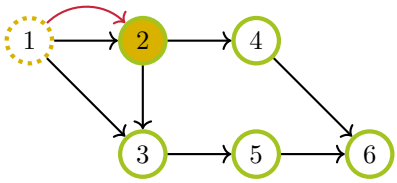
(入力の形式は例題 “Graph” と同じ)

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_11_B

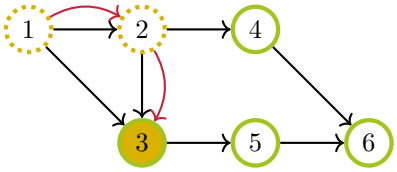
深さ優先探索(参考書 [3, p. 33]) は，全ての節点を訪問する手法の一つで，現在訪問中の頂点に近い頂点から順に訪問する．まずは再帰的手続きを用いた実装を紹介する．



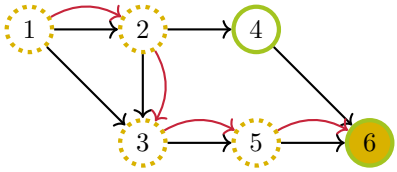
時刻 1: 頂点 1 から開始



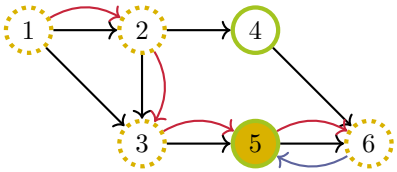
時刻 2: 行き先候補 2 と 3 から 2 を選択



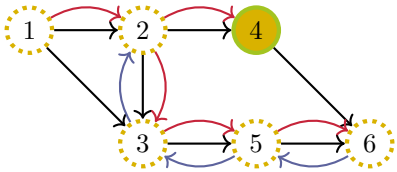
時刻 3: 同様に 3 を選択



時刻 5: 同様に 6 まで進むと行き先がなく



時刻 7: 親に戻る



時刻 9: 未訪問の行き先があれば進む

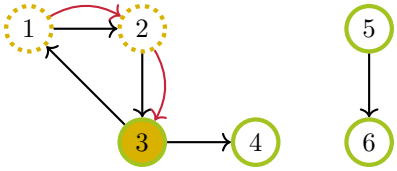
アルゴリズムとして，dfs(int cur) を cur (current node の略) 以下のノードを全て訪問する手続きと定義する．そのような手続きを実装するためには，子節点について再帰的に dfs を呼べば良い．

C++

```
1  int time = 0
2  void dfs(int cur) { // cur を訪問
3      // cur の訪問時刻を記録
4      time += 1;
5      // 動作確認用表示
6      cerr << "visiting␣" << cur << '␣' << time << endl;
```

```
7
8       for (dst ...) { // 全ての節点 dst について
9           if (...) { // cur から dst に辺があり，dst を未訪問なら
10              dfs(dst)
11          }
12      }
13      // cur の訪問終了時刻を記録
14      time += 1;
15      // 関数の終わりで親に戻る (青矢印)
16  }
```

ループの検出と非連結のグラフ



どのようなグラフを対象とするかは予め想定する必要があり，この問題では上記のようなグラフも与えられうる．まず頂点 3 まで進んだ時点で，頂点 1 に進まないように注意しよう (無限ループになる)．防ぐためには，行き先候補の訪問時刻を確認して既に尋ねたことがあるかどうかを識別すると良い．このような，訪問済の頂点に戻る辺を後退辺back edge と言う場合がある．

またこの問題では，頂点 1 を出発点にしての DFS を終えたら，次は頂点 5 を出発点にしての DFS を始め，すべての頂点を訪問するまで続けることが求められる．DFS を終えたら，頂点番号を増やしながら未訪問の頂点がないかを確認し，見つかればそこを出発点に DFS を行えば良い．

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

8.3.1 章ではこれを幅優先探索で解くように紹介した．今回は深さ優先探索で解いてみよう．

10.2.1 様々な問題

問題	**Map of Ninja House⋆**	(アジア地区予選 2002)

探索履歴から地図を復元せよ

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1236

参考資料: http://hos.ac/slides/20110504_graph.pdf 28 枚目くらいから

問題	**Articulation Points**⋆	(AOJ)

関節点を求めよ.
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_3_A&lang=jp

根についてと，根以外の頂点についてそれぞれ判定する.

後者の例:各頂点を訪問する際に，根からの深さ (距離; たどった辺の数) を記録する．探索中に back edge を見つけた場合は，辺の先の頂点の深さを見て，その頂点がどこまで上に行けるかを管理する．各親子について，親の深さと子がどこまで上に上がれるかを元に判定することができる.

問題	**TELE**⋆	(Croatia OI 2002 Final Exam - Second Day)

(意訳) 放送局が番組を配信する計画を立てる．受信できた場合に視聴者が払う額はあらかじめ与えられる．配信路は木構造になっている．木の節点は，根が放送局自身，葉が視聴者．他が中継装置である．赤字にならずに配信できる人数は何人か?

(筆者注) 各コスト C の制約がないが，正で 5000 以下の模様.

http://poj.org/problem?id=1155

木と動的計画法: 節点 i から j 人に配信するコスト T[i][j] のようなものを管理しながら根から深さ優先探索を行う．各節点で，たとえば二つの子を持つ内部節点で 3 人に配信する場合の収益は、(0,3), (1,2),... (3,0) のようなすべての割り当ての中から最大を選ぶ必要がある.

10.3　再帰下降と構文解析: 四則演算の作成

一定の文法規則に従って作成されている文字列から，文法の構成要素を読み取って，構造を把握したい．たとえば，四則演算の式を表す文字列を与えられて，実際の計算を行いたい．その際には，「文字列の指定位置から，ある文法規則を満たす要素を読み取り返り値とし，その分だけ位置を進める」というような，再帰的関数と手続きの複合させた関数を考えると便利である.

10.3.1　足し算を作ってみよう

■大域変数　:

C++
```
1 const string S = "12+3";
2 size_t cur = 0; // 解析開始位置 cursor の略記
3 int parse();
```

実行例: (以下のように動作するものをこれから作る)

C++
```
1  int main() {
2    int a = parse();
3    assert(a == 15);
4    assert(cur == S.size());
5  }
```

■1 文字読む関数の準備

C++
```
1  // 1 文字読んで cur を進める
2  char readchar() {
3    assert(cur < S.size());
4    char ret = S[cur];
5    cur += 1;
6    return ret;
7    // return S[cur++]; と一行で書くこともできる
8  }
9  char peek() { // 1 文字読むが cur を進めない
10   assert(cur < S.size());
11   return S[cur];
12 }
```

■assert って何? (復習)　ソースコード

C++
```
1  #include <cassert>
2  int factorial(int n) {
3    assert(n > 0); // (*)
4    if (n == 1) return 1;
5    return n * factorial(n-1);
6  }
```

実行例

C++
```
1  cout << factorial(3) << endl; // 6 を表示
2  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'
```

C++

```cpp
#include <cctype>
int digit() {
  assert(isdigit(peek()));  // S[cur] が数字であることを確認
  int n = readchar() - '0'; // '0' を 0 に変換
  return n;
}
```

文法通りに実装する (Number)

```
Number := Digit の繰り返し
```

C++

```cpp
int number() {
  int n = digit();
  while (cur < S.size() && isdigit(peek())) // 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
```

C++

```cpp
int expression() {
  int a = number();
  char op = readchar();
  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:

C++

```cpp
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

C++
```
1  if (op == '+') return a + b;
2  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

C++
```
1  int expression() {
2      int sum = number();
3      while (peek() == '+' || peek() == '-') { // 足し算か引き算が続く間
4          char op = readchar();
5          int b = number();
6          if (op == '+') sum に b をたす;
7          else sum から b を引く;
8      }
9      return sum;
10 }
```

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

10.3.2 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 }
```

C++

```
1  int term() {
2    int a = number();
3    while (cur < S.size()
4           && (peek() == '*' || peek() == '/')) {
5      char op = readchar();
6      int b = number();
7      if (op == '*') a *= b; else a /= b;
8    }
9    return a;
10 }
```

四則演算の実装 (Expression)

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

C++

```
1  int expression() {
2    int a = term();
3    while (cur < S.size())
4           && (peek() == '+' || peek() == '-')) {
5      char op = readchar();
6      int b = term();
7      if (op == '+') a += b; else a -= b;
8    }
9    return a;
10 }
```

10.3.3 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:

C++

```
1  int expression(); // 前方宣言
2  int factor() {
3    if (peek() != '(') return number();
4    cur += 1;
5    int n = expression();
6    assert(peek() == ')');
7    cur += 1;
8    return n;
9  }
```

term() の実装も，文法に合わせて調整すること.

10.3.4 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)

10.3.5 練習問題

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&

(問題 続き)

lang=jp

回答例:

C++

```
1  /*const*/ string S; // 値を変更するので const 属性を削除
2  ...
3  int main() {
4      int N;
5      cin >> N;
6      for (int i=0; i<N; ++i) {
7          cur = 0;
8          cin >> S;
9          S.resize(S.size()-1); // 最後の=を無視
10         cout << expression() << endl;
11     }
12 }
```

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

回答例:

1. 変数がなければ式の値を求めることは簡単である．すなわち，文字列内部の P, Q, R をそれぞれ 0,1,2 に置換してから構文解析して求めれば良い．P, Q, R への値の割り当ては 3^3 通りあるので，それだけ繰り返す.
2. (特に C++11 でお勧め)P, Q, R への値の割り当てを整数の配列 `int a[3]` で表すとする．割り当て `a` を引数に取り，その割り当てでの式の値を返す関数を構文解析で作成する．構文解析結果を活用する部分は以下のようになる:

C++11

```
1  int solve() {
2      cur = 0;
3      auto tree = parse();
4      int count = 0;
5      for (int p:{0,1,2})
6          for (int q:{0,1,2})
7              for (int r:{0,1,2}) {
8                  int a[] = {p, q, r};
9                  if (tree(a) == 2) ++count;
```

```
10              }
11      return count;
12  }
```

割り当てに対する式の値を返す関数は細かい関数を組み合わせて作る．たとえば文字 c が数を表す場合, 割り当てによらず同じ値を返す定数関数を作成する return [=](int[3]){ return c-'0'; };．アルファベットだった場合は, 引数に応じた値を返すので return [=](int a[3]){ return a[c-'P']; }; というようにすれば良い．括弧でくくられた二項演算子の場合，四則演算の時と同じように左側と右側を解析した関数を left, right などと作成したうえで return [=](int a[3]) { return min(left(a), right(a)); }; のように左右の関数を呼び出す関数を作成する．

C++11

```
1  #include <functional>
2  typedef function<int(int[3])> node_t;
3  node_t parse() {
4      char c = readchar();
5      if (isdigit(c))
6          return [=](int[3]){ return c-'0'; };
7      if (isalpha(c))
8          return [=](int a[3]){ return a[c-'P']; };
9      node_t left = parse();
10     if (c == '-')  // left(a) に対して'-' の演算を行う
11         return [=](int a[3]) { return ...; };
12     assert(c == '(');
13     char op = readchar();
14     node_t right = parse();
15     ++cur; // ')'
16     // 以下二項演算子なので left(a) と right(a) に対して..
17     if (op == '*')
18         return [=](int a[3]) { return ...; }; // '*' の演算を行う
19     return [=](int a[3]) { return ...; }; // 同'+' の演算を行う
20 }
```

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．

問題	**Chemist's Math**	(アジア地区予選 2009)

与えられた反応に必要な各物質の比を求める．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1300

回答例: 各分子の構成を調べて，連立方程式を解く．

10.4 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する．

コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である．

C++

```
1 // minmaxsum.cc
2 #if 0
3   所要時間は..であった．方針は....
4   ...
5 #endif
6 int main() {
7 }
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- “Depth First Search” と “Smart Calculator” を含む，2 題以上を解く
- (“Map of Ninja House” または “如何に汝を満足せしめむ？”) を含む，2 題以上を解く

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

第 11 章

平面の幾何

2018-12-12 Wed

概要

さまざまな状況で計算機で図形的問題を扱う機会がある．ここでは幾何の問題を扱う基本を紹介したい (詳しく学ぶには専門の授業を受講されたい)．たとえば，平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．人間には簡単な図形の概念でも，プログラムとして記述する場合は直感的ではない方法が適している場合がある．また浮動小数 (`double` など) を扱うので，誤差に注意する必要も生ずる．

11.1 浮動小数と誤差 (floating point numbers and errors)

計算機内の数値データは，有限長のビット列で表現される．

例: 整数の場合 int (演習環境では 32bit), long long (演習環境では 64bit).

整数を，小数を表すために用いることもできる．たとえば，小数点以下 2 桁のみ扱う場合は (円に対する銭など)，数値を 100 倍すれば整数になる．その場合には，内部では整数として演算を行い，表示する場合のみ，`cout << x/100 << "." << (x%100) << endl;` などと小数表記にする方法がある．(固定小数点)

11.1.1 浮動小数

より大きな数や小さな数を柔軟に表現するためには，符号部 (sign)，指数部 (exponent)，仮数部かすうぶ (mantissa) を持つ実数表現 (浮動小数点表現) が利用される．

IEEE 754 倍精度 `double` (64bit) では，符号部 s，指数部 e，仮数部 m の 3 つの整数の組で，次のような一つの数を表す:

$$(-1)^s(1+m\cdot 2^{-52}))\cdot 2^{(e-1023)}. \tag{11.1}$$

全体を 64bit で表現し，符号部 s には第 0 ビット，指数部 e 第 1–11 ビット (11bits)，仮数部 m 12–63 ビット (52bits) を割り当てる．

次のプログラムは共用体 (`union`) の機能を利用して，`double` 型の内部表現を表示する．なお `bitset` は，ここでは整数の二進表現を得るためだけに借用している．

C++

```cpp
#include <iostream>
#include <bitset>
using namespace std;
union double_long_long {
  double floating_value;
  unsigned long long integer_value;
};
void show_double(double v) {
  double_long_long u;
  u.floating_value = v;
  cout << v << " = " << bitset<64>(u.integer_value) << endl;
  long long sign = u.integer_value >> 63;
  long long exponent = (u.integer_value >> 52)& ((1<<11)-1);
  long long mantissa = u.integer_value & ((1ul<<52)-1);
  cout << " sign = " << sign << endl;
  cout << " exponent = " << exponent << ' ' << bitset<11>(exponent) << endl;
  cout << " mantissa = " << bitset<52>(mantissa) << endl;
  cout << " => " << (sign ? "-" : "+")
       << " (1+" << (double)mantissa/(1ul<<52)
       << ") * " << "2^(" << (exponent-1023) << ")" << endl;
}
int main() {
  show_double(1.0);
  show_double(-1.0);
  show_double(0.5);
  show_double(0.25);
  show_double(0.75);
}
```

実行例: 式 11.1 と照らして確認せよ.

```
1 = 0011111111110000000000000000000000000000000000000000(略)
 sign = 0
 exponent = 1023 01111111111
 mantissa = 0000000000000000000000000000000000000000000000000(略)
 => + (1+0) * 2^(0)
-1 = 1011111111110000000000000000000000000000000000000000(略)
 sign = 1
 exponent = 1023 01111111111
 mantissa = 0000000000000000000000000000000000000000000000000(略)
 => - (1+0) * 2^(0)
0.5 = 0011111111100000000000000000000000000000000000000000(略)
 sign = 0
 exponent = 1022 01111111110
 mantissa = 0000000000000000000000000000000000000000000000000(略)
 => + (1+0) * 2^(-1)
0.25 = 0011111111010000000000000000000000000000000000000000(略)
 sign = 0
```

```
 exponent = 1021 01111111101                                                    18
 mantissa = 0000000000000000000000000000000000000000000000000(略)              19
 => + (1+0) * 2^(-2)                                                            20
0.75 = 0011111111101000000000000000000000000000000000000000(略)                21
 sign = 0                                                                       22
 exponent = 1022 01111111110                                                    23
 mantissa = 1000000000000000000000000000000000000000000000000(略)              24
 => + (1+0.5) * 2^(-1)                                                          25
```

練習: 0.875 $(1/2+1/4+1/8)$ を double で表現した際のビット列を予想せよ．上記のプログラムを用いて確認せよ．

11.1.2 様々な誤差

丸め誤差 (round-off errors) 小数を 2 進数で表現する場合，有限桁では近似的にしか表せないことに起因する誤差 (10 進の小数で $1/3$ を有限桁では正確に表現できないことに相当)

例 $0.1_{(10)} = 2^{-4}+2^{-5}+2^{-8}+2^{-9}\ldots = 1.10011\ldots_{(2)} \cdot 2^{-4}$

練習: 次のプログラムの出力を予想しよう?

C++

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4   double a = 0.1;
5   double b = 0.3;
6   if (a*3 == b) cout << "OK" << endl;
7   else cout << "NG" << endl;
8 }
```

```
0.1*3 = 00111111110100110011001100110011001100110011001100110011001100110100
  0.3 = 00111111110100110011001100110011001100110011001100110011001100110011
```

桁落ち (loss of significance) 有限の有効桁数で表現されている数について，ほぼ同じような数値の差を取ると有効桁数が減少すること

例: $0.124 - 0.123 = 0.001$

注:元々誤差なく表現されている数同士の演算では発生しない．

情報落ち (information loss) 大きさの異なる数値の加減算では、小さな数値は大きな数値の有効範囲外になり無視されてしまうこと

例: $0.124 + 0.0000000123 = 0.124$

打ち切り誤差 (truncation errors) 関数の値を無限級数を用いて数値計算をする場合，有限の項数で打ち切って近似することにより生ずる誤差

問題　**情報科学 2014 年試験問題**　(情報図形部会)

情報科学 2014 年試験問題の問題 3 の小問 1-6 に回答せよ (小問は 7,8 は不要).

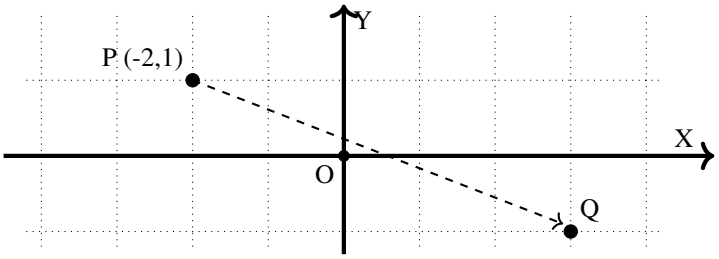
小問 2 の出題意図補足: 直前で `1/a+1` が `1.001` と表示されていることと整合する説明を行うこと.

小問 3 の出題意図補足: (c) のほうが (b) よりも誤差が大きいことを説明してほしい

http://lecture.ecc.u-tokyo.ac.jp/johzu/joho-kagaku/common-exam-2014.pdf

11.2 概要: 点の表現と演算

平面上の点を C++ で表現する場合に，以下のように複素数で表現すると，標準ライブラリを様々な演算に活用出来る (実部 `real()` が `x` で，虚部 `imag()` を `y` に対応させる):



C++

```
1  #include <complex>
2  #include <cmath>
3  typedef complex<double> xy_t;
4  xy_t P(-2, 1); // 初期化
5  cout << P << endl; // (debug 用) 表示
6  cout << P.real() << endl; // x 座標
7  cout << P.imag() << endl; // y 座標
8  cout << abs(P) << endl; // ベクトル OP の長さ (O は原点)
9  cout << norm(P) << endl; // norm(P) = abs(P)^2
10 xy_t Q = P + xy_t(5, -2); // 点 Q は点 P を (5,-2) だけ平行移動した位置とする
11 Q *= xy_t(cos(a), sin(a)); // 点 Q を原点を中心に a (ラジアン) だけ回転
```

C (gcc) の場合

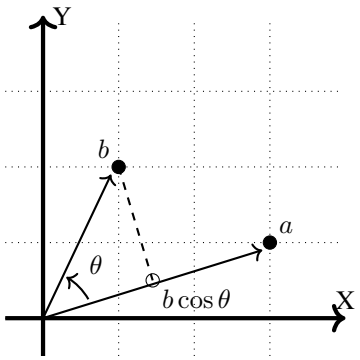
C

```
1  #include <complex.h>
2  #include <math.h>
3  complex a = 0.0 + 1.0I; // 初期化
4  complex b = cos(3.14/4) + sin(3.14/4)*I;
5  printf("%f␣%f\n", creal(a), cimag(a)); // 実部，虚部
6  a *= b; // 乗算
7  printf("%f␣%f\n", creal(a), cimag(a));
```

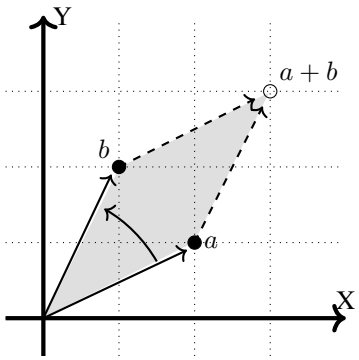
乗算記号の入れ忘れに注意

数の 5 と変数 k の積を求める場合は，5*k と書き，5k と書くとコンパイルエラーになる．ところが，文字 i,j,I,J に関しては，5j などの表記が上記の虚数と解釈され，コンパイルエラーにはならない．cout に表示すると，bool にキャストされて 1 と表示される．知らないと見つけにくい，と思われる．

よく使う演算



(1) 内積: $|a||b|\cos\theta$



(2) クロス (外) 積: 網掛け部分の符号付き面積

C++

```
1 // 図 (1) 内積:  a.x*b.x +a.y*b.y (|a||b|cosθ に一致)
2 double dot_product(xy_t a, xy_t b) { return (conj(a)*b).real(); }
3 // 図 (2) クロス (外) 積:  a.x*b.y - b.x*a.y
4 // ベクトル a,b が作る三角形の符号付き面積の二倍
5 double cross_product(xy_t a, xy_t b) { return (conj(a)*b).imag(); }
6 // (図なし) 射影:  原点と p を結ぶ直線に b を射影
7 xy_t projection(xy_t p, xy_t b) { return b*dot_product(p,b)/norm(b); }
```

三角形の符号付き面積の紹介が本章前半の主要なテーマである．これは原点と点 a,b が作る三角形の面積を，符号付きで求める．符号は，原点と点 a,b がこの順で反時計回りの位置関係にある場合に正，時計回りの場合に負となる．面積だけでなく，これから見るように向きの判定にも用いられる．

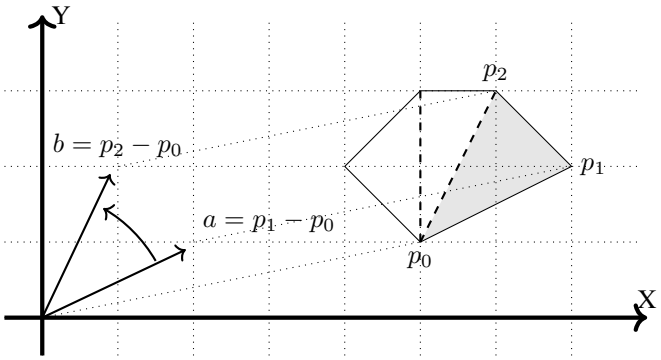
内積は，直線上に点を射影する際に便利である．

11.3　三角形の符号付き面積の利用

例題　**Area of Polygon**　(PC 甲子園 2005)

凸多角形の面積を求めよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0079



(単純) 多角形は三角形に分解できるので，三角形の面積が計算できれば，多角形の面積が計算できる．特に凸多角形の場合は，一つの頂点とその頂点を含まない辺が構成する三角形で綺麗に分割できる．

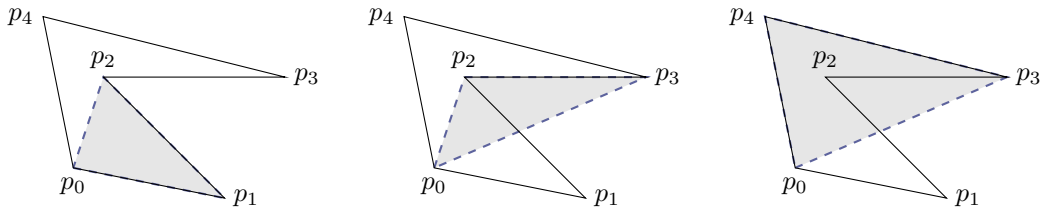
C++

```
1  xy_t P[110];
2  int main() {
3    // 入力例 読み込んだ点の個数を N とする
4    int N=0;
5    double x, y;
6    while (scanf("%lf,%lf", &x, &y) != EOF) {
7      P[N++] = xy_t(x,y);
8    }
9    // 面積計算
10   double sum = 0.0;
11   for (int i=0; i+2<N; ++i) {
12     xy_t a=P[0], b=P[i+1], c=P[i+2];
13     sum += ... // 三角形 abc の面積を合計
14   }
15   printf("%.6f\n", abs(sum));
16 }
```

例題 **Polygon - Area** (AOJ)

単純多角形の面積を計算する．凸とは限らないが，頂点は反時計回りで与えられる．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_3_A&lang=jp
(類題: Area of Polygons (国内予選 1998) http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1100 頂点が与えられる向きのみ異なる)

凸でない単純多角形に対して，先ほどと同様の分割を行うと三角形に重なりが生ずるが，ここで符号付き面積の符号も含めて合計すると，(不思議なことに) 正しい面積を得られる．多角形の頂点は反時計回りに順に与えられている必要がある．なお分割の中心として p_0 を使ったが，任意の点 (たとえば原点) と各辺の作る三角形を考えても良い．



$p_0p_1p_2$ (sgn: +)　　$p_0p_2p_3$ (sgn: -)　　$p_0p_3p_4$ (sgn: +)

11.3.1 平行の判定

例題 **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

C++

```
1  const double eps = 1e-11;
2  double x[4], y[4];
3  int N;
4  int main() {
5      cin >> N; // 問題数
6      for (int t=0; t<N; ++t) {
7          for (int i=0; i<4; ++i)
8              cin >> x[i] >> y[i]; // x0,y0..x3,y3
9          xy_t a[2] = {
10             xy_t(x[0],y[0]) - xy_t(x[1],y[1]),
11             xy_t(x[2],y[2]) - xy_t(x[3],y[3])
12         };
13         bool p = abs(a[0]とa[1]の符号付き面積) < eps;
14         cout << (p ? "YES" : "NO") << endl;
15     }
16 }
```

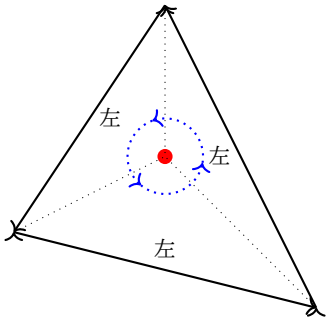
補足: 向きや角度の判定には，sin や arg などのライブラリ関数を用いることもできるが，計算誤差の観点から可能な限り符号付き面積で計算するほうが良い．たとえば三角関数の汎用的な実装方法では Taylor 展開を用いる．[*1]

■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

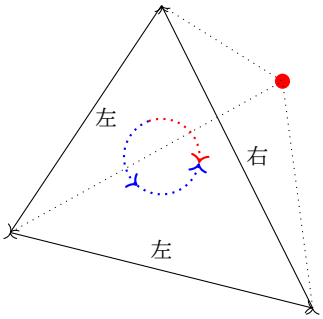
[*1] 参考: FreeBSD の実装 https://svnweb.freebsd.org/base/release/10.1.0/lib/msun/src/k_cos.c?view=markup

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 で表した時の丸め誤差，400 は $|(a+\varepsilon)(d+\varepsilon) - (b+\varepsilon)(c+\varepsilon)|$ 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)．なお，環境依存になるが比較的新しい Intel や AMD の CPU と比較的新しい gcc を用いる場合は，long double や__float128 などを用いることで 80 bit や 128 bit というより良い精度で演算することもできる．それぞれ注意点があるので，使用する場合は文献を調査のこと．

11.3.2 内外判定



点が図形の内部



点が図形の外部

例題 **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．

C++

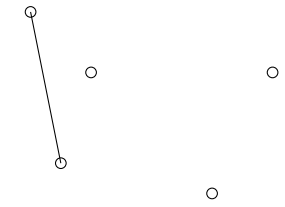
```
1  double x[4], y[4];
2      while (true) {
3          for (int i=0; i<4; ++i) cin >> x[i] >> y[i];
4          if (!cin) break;
5          xy_t a(x[0],y[0]), b(x[1],y[1]), c(x[2],y[2]), p(x[3],y[3]);
6          // pab の符号付き面積の 2 倍は，cross_product(a-p,b-p)
7          // pbc の符号付き面積の 2 倍は，cross_product(b-p,c-p)
8          // pca の符号付き面積の 2 倍は，cross_product(c-p,a-p)
9          bool ok = 符号が揃っている
10         cout << (ok ? "YES" : "NO") << endl;
11     }
```

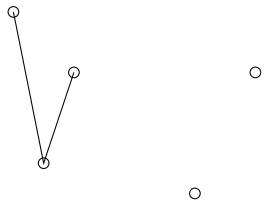
11.3.3 凸包

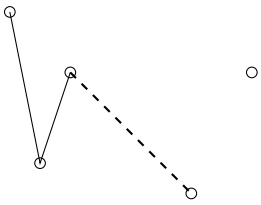
問題	**Convex Polygon - Convex Hull⋆**	(AOJ)

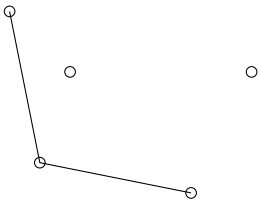
与えられた点の集合の凸包を求めよ．凸包が何かは類題の図を参照．
(注: 辺上の点を出力させる，少し特殊な設定である)
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_4_A&lang=jp
(類題: Enclose Pins with a Rubber Band (PC 甲子園 2004) http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0068 こちらの方が素直な設定)

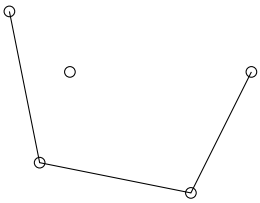
回答例: 点を X 座標でソートし，最小値 (左端の点) から順に右に伸ばして，まずは下半分の外周を構成する．凸包の下側の外周は，どの連続する 3 点をとっても左向きである．もし構成中に，直近の 3 点が右向きになってしまったら，(本来凸包に含まれない点を入れてしまった状況であるので) 向きが正しくなるまで直近の点を取り除く．上半分も右端から同様に行う．点の個数を N とすると，上記の半周を求める手続きは $O(N)$ でできるため，ソートに要する時間を含めて全体の計算時間は $O(N \log N)$.


t=0: 左端 (x 座標最小) から


t=1: 順に線をつなぐ


t=2: 右向きになったら


t=4: 必要なだけ点を削除してつなげ直し


t=5: 下側完成

11.4 応用問題

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

(問題 続き)

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 [3, p. 229])
ヒント: 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．以下白い文字で

問題 **Chain-Confined Path**⋆ (国内予選 2012)

円環を通る最短経路を求めよ．
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1183&lang=jp

ヒント: 最短経路の形状を考える
解法:

問題 **Neko's Treasure**⋆ (模擬地区予選 2009)

壁をいくつ乗り越えるかを求める (問題文参照)
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2181

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話の枕．
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．https://szkopul.edu.pl/problemset/problem/4zZWWzI9tb_9FGiIbDgtFPUP/site/

11.5 今週の課題

問題を解いて，AOJ で Accepted を得たことを確認して，各問題指定の考察課題もしくは指定がない問題については感想 (学んだことや課題にかけた時間など) とともに，ITC-LMS に提出する．

コメントの記述は，`#if 0` と`#endif` で囲むことが簡便である．

C++

```
1 // minmaxsum.cc
2 #if 0
3   所要時間は..であった．方針は....
4   ...
5 #endif
6 int main() {
7 }
```

扱う問題は以下のどちらかで合格 (言語指定なし):

- 例題 “Area of Polygons” と問題「情報科学試験問題」を解いて提出．情報科学試験問題は，満点を得られるまで再提出すること．
- 応用問題から “Convex Hull” を含む 2 題以上提出する． 感想 *(学んだことや課題にかけた時間など)*を添えること．

提出先: ITC-LMS
提出の注意:

- 問題毎にテキストファイルを一つ作り，提出する (*zip* は避ける; PDF, Microsoft Word, rtf などは避けること)．グラフを作成した場合は，グラフのみ別ファイルで提出する．
- 提出物は履修者全体に公開される．この授業では，他者のソースコードを読んで勉強することを推奨するが，課題を解く上で参考にした場合は，何をどのように参加したかを明記すること．
- 締切: 授業終了時，次回授業の前日，学期終了時のそれぞれで最善のものを提出する

第 12 章

補間と数値積分

2018-12-19 Wed

事務連絡

- この授業の成績は，各回の提出物で評価する．試験は行わない．
- ITC-LMS 内の授業アンケートに回答すること

概要

数学に関連のあるプログラミングについて紹介する．概念の都合から， Lagrange 補間を先に説明するが，プログラミングとしては数値積分を (夏学期に触れていない人は「例題」から) 体験しておくことを勧める．

12.1　浮動少数点数の補足

0.1 刻みで 0 から 0.9 まで処理する課題を考える．この時，ループカウンタには整数を用いることが定跡である．

良い実装

Python3

```
1 i = 0
2 while i < 10:
3   print(i/10.0)
4   i += 1
```

```
0.0
0.1
(中略)
0.8
0.9  # 全 10 行
```

悪い実装 (0.1 から...11 行表示)

Python3

```
1 i = 0.0
2 while i < 1.0:
3   print(i)
4   i += 0.1
```

```
0.0
0.1
(中略)
0.7999999999999999
0.8999999999999999
0.9999999999999999      # 全 11 行
```

12.2 Lagrange 補間多項式

3 点 $(a, v_a), (b, v_b), (c, v_c)$ を通る 2 次の多項式は次のように一意に定まる:

$$L(x) = \frac{(x-b)(x-c)}{(a-b)(a-c)}v_a + \frac{(x-a)(x-c)}{(b-a)(b-c)}v_b + \frac{(x-a)(x-b)}{(c-a)(c-b)}v_c. \tag{12.1}$$

一般に，与えられた N 個の点を通る次数最低の多項式は一意に定まる:

$$L(x) = \sum_{k=0}^{N-1}\left(\prod_{i\neq k}\frac{(x-x_i)}{(x_k-x_i)}\right)\cdot v_{x_k}.$$

(この式でくらっとした場合は，以下を無視して 12.3 節へ進んで良い．)

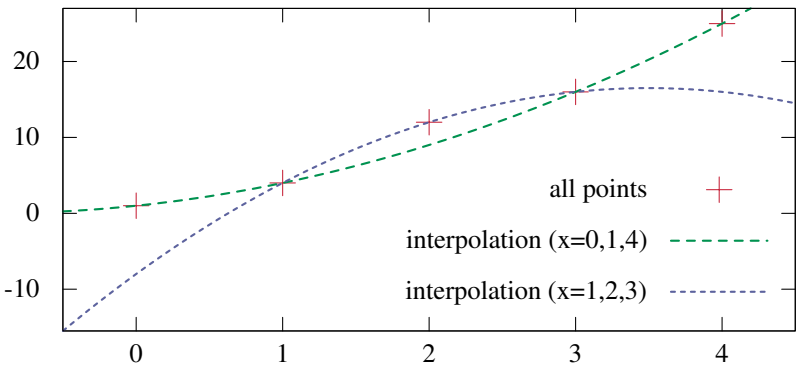
問題	**Find the Outlier**	(アジア地区予選 2012)

秘密の多項式 $f(x)$ の次数 d と $d+3$ 個の点，$(0, v_0), (1, v_1), \cdots, (d+2, v_{d+2})$ が与えられる．点 (i, v_i) は $f(i) = v_i$ を満たすが，例外として一点 $(i', v_{i'})$ のみ $v_{i'}$ に 1.0 以上の誤りを含む．その点 i' を求めよ．入力は 10^{-6} より大きな誤差を含まない．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1328

方針:

- $d+1$ 個の点を選んで Lagrange 補間多項式を作る．その式の値と，選ばれなかった点が一致するかを考える．
- i' を除いた $d+2$ 個の点についてだけ考えると，どの $d+1$ 個の点で作られた補間多項式も f と一致し，残りの 1 点を通る．
- 点 i' を含む $d+2$ 個の点については，同様の補間多項式が残りの 1 つの点と一致しないような ($d+1$ 個と 1 個) の点の選び方がある．



1 つ目のサンプル入力の例: outlier である (2,12) を外した 3 点で補間するとどの場合も同じ関数が得られ，outlier の 1 点は外れるが，補間に使用しなかった残りの一点を通る (緑破線)．outlier である (2,12) を含む 3

点で補間すると，残りの 2 点を外れる (青点線).

点 $f(n)$ を，点 (n, v_n) と (E, v_E) を除いた $d+1$ 個の点から求める関数 `interpolate(n,E)` は以下のように作ることができる.

C++

```cpp
int D;
double V[16];
double interpolate(int n, int E) {
    // 補間多項式 L(x) の x=n での値 L(n) を計算する
    // L(x) の作成において，点 (n,V[n]) と (E,V[E]) は使わない
    double sum = 0.0;
    for (int k=0; k<D+3; ++k) {
        if (/*k が n か E なら*/) continue;
        double p = V[k];
        for (int i=0; i<D+3; ++i)
            if (! (/*i が k でも n でも E でもなければ*/))
                p *= (...-i)/(double)(k-i);
        sum += p;
    }
    return sum;
}
```

Python3

```python
# 配列 V に値が入っているとする
def interpolate(n, e):
  total = 0.0
  for k in range(len(V)):
    if ... # k が n か E なら
        continue
    p = V[k]
    for i in range(len(V)):
        if  ... # i が k でも n でも E でもなければ
            p *= 1.0*(n-i)/(k-i)
    total += p
  return total
```

この関数 `interpolate` を用いると，点 (E, V_E) が異常だったと仮定して，残りの $d+2$ 個の点に異常がないか調べる関数 `outlier(E)` を次のように簡単に作ることが出来る.

C++

```cpp
bool outlier(int E) {
    for (int i=0; i<D+3; ++i) {
        if (i==E) continue;
        double p = interpolate(i, E);
        if (/*pとV[i]の絶対値(abs)が0.1より離れていたら*/)
            return false;
    }
    return true;
}
```

Python3

```python
def outlier(e):
    for i in range(len(V)):
        if i == e:
            continue
        p = interpolate(i, e)
        if ... # p と V[i] の絶対値 (abs) が 0.1 より離れていたら
            return False
    return True
```

C++

```cpp
#include<iostream>
#include<cmath>
// この辺に上記の関数を定義する
int main() {
    while (cin >> D && D) {
        for (int i=0; i<D+3; ++i) cin >> V[i];
        for (int i=0; i<D+3; ++i)
            if (outlier(i)) { // i を無視すれば全て整合する
                cout << i << endl;
                break;
            }
    }
}
```

Python3

```python
# global 変数 V と上記の関数を事前に定義しておく
while True:
  d = int(input())
  if d == 0:
      break
  V = [float(input()) for _ in range(d+3)]
  for i in range(d+3):
      if outlier(i): # i を無視すれば全て整合する
          print(i)
          break
```

12.3 数値積分 (区分求積法) とシンプソン公式

例題 **Integral** ()

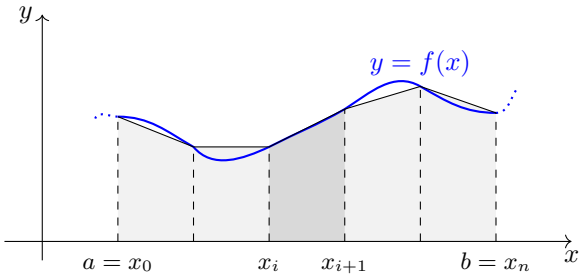
長方形の和を用いて面積の近似値を求めよ.

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0014&lang=jp

関数 $f(x)$ の区間 $[a,b]$ の定積分 $\int_a^b f(x)dx$ を求めたい．解析的に解く方法以外に，数値計算で近似値を求める種々の方法が存在する．もっとも簡便な方法は，区間を細かく分けて長方形の面積の和で近似するものである．(例題の話題はここで区切り)

12.3.1 台形公式

小分けにした短冊部分について，長方形を台形に変更すると，分割数に対応する精度を高めることができる．分割した区間の幅を h とすると誤差は $O(h)$ から $O(h^2)$ に改善する (計算誤差の影響を無視すれば，刻みを 10 倍細かく取ると，精度が 100 倍改善すると予想できる)．



応用

実際には数値誤差の影響で h を小さくしても，ある程度で誤差は減少しなくなる．各項の丸め誤差が $\epsilon = 10^{-16}$ と予想される時，適切な h はどのように見積もれば良いか?

12.3.2 シンプソン公式

シンプソン公式 (Simpson's rule) では，直線の代わりに二次式で近似し，精度は $O(h^4)$ に改善する．先ほどまでと記法が異なるが $a = x_i$, $b = x_{i+1}$ として，区間内の三点 $(a, f(a)), (\frac{a+b}{2}, f(\frac{a+b}{2})), (b, f(b))$ を使った補間多項式 (式 (12.1)) で表現して整理すると，その区間の面積は以下のように近似できる:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right).$$

もし区間内で $f(x)$ が二次以下の多項式であれば，計算誤差を無視すれば正確な値を得ることが出来る．

問題　One　(立命館合宿 2013)

窓枠内に見えている山々と空との境界の長さを求めよ．入力の制約から山の頂点は窓枠内 (境界含む) に存在する．

http://judge.u-aizu.ac.jp/onlinejudge/cdescription.jsp?cid=RitsCamp13Day2&pid=I

注: 区間 $x \in [a,b]$ の $f(x)$ の周長は，$f(x)$ の導関数を $f'(x)$ として $\int_a^b \sqrt{1+f'(x)^2}dx$ で与えられる．
山と空の境界をどの放物線が与えるかは，場所によって異なる．そこで x 座標を用いて区間を分割し，各区

間毎の周長を合計すると良い．分割方法は，各放物線同士の交点 (あれば) や窓の下端との交点 (あれば) の x 座標を列挙し，それらを用いて窓内の空間を分割すれば十分かつ，区間の数も問題ない範囲である．各区間ではまったく山が見えないもしくは，一つの放物線が空との境界をなす．どの放物線が一番上にあるかは，区間の中央の x での高さで比較すれば求まる．積分は，数値計算がお勧め．

問題	**Intersection of Two Prisms**	(ACM-ICPC 東京大会 2010)

z 軸方向に無限の高さを持つ多角柱 P1 と，y 軸方向に無限の高さを持つ多角柱 P2 の共通部分の体積を求めよ．

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1313

方針 (参考書 [3, pp. 236–]): 求める立体を平面 $x = a$ で切断した断面を考える．断面は面積を持たないか，または y 軸と z 軸に平行な長方形になる．つまり，断面の面積は x の関数 $f(x)$ と表現できるので，その面積を x 軸方向に積分すると体積 $\int_x f(x)dx$ を得られる．なお積分区間は立体の頂点の存在する x 座標で区切る．

ある x 座標での切断面での y 軸または x 軸の幅を求める関数 `width(polygon, x)` は，たとえば以下のように与えられた多角形の辺を一巡して求めることができる．

Python3

```
def width(polygon, x): # polygon に x-y または x-z 平面の座標が順に入っている
    w = []
    for i in range(len(polygon)):
        p, q = polygon[i], polygon[(i+1)%len(polygon)]
        if p[0] == x:
            w.append(p[1])
        elif (p[0] < x and x < q[0]) or (p[0] > x and x > q[0]):
            x0, y0 = p[0], p[1]
            x1, y1 = q[0], q[1]
            w.append(y0 + 1.0*(y1-y0)*(x-x0)/(x1-x0))
    assert len(w) > 0
    return max(w) - min(w)
```

この関数を利用して，体積を求める関数 `volume` を，たとえば以下のように作成できる．なお `Pxy` は多角形 P1 の (x,y) 座標を順に格納した配列，`Pxz` を多角形 P2 の (x,z) 座標を順に格納した配列, `X` を P1 と P2 に登場する X 座標全を昇順に格納した配列とする．

Python3

```
def volume(Pxy,Pxz,X):
    X = sorted(set(X))
    total = 0.0
    xmin = max(min(Pxy)[0], min(Pxz)[0])
    xmax = min(max(Pxy)[0], max(Pxz)[0])
    for i in range(len(X)-1):
        a, b = X[i], X[i+1]
        if not (xmin <= a and a <= xmax and xmin <= b and b <= xmax):
            continue
        m = (a+b)/2.0
```

```
11          va = width(Pxy, a)*width(Pxz, a)
12          vb = width(Pxy, b)*width(Pxz, b)
13          vm = width(Pxy, m)*width(Pxz, m)
14          area = ... # (a,va), (m,vm), (b,vb) を通る区間 [a,b] の定積分
15          total += area
16      return total
```

12.4 今週の課題

問題を 1 題解いて提出する (“Intersection of Two Prisms” がお勧め)

提出先: ITC-LMS

第 13 章

予備

2019-01-09 Wed

(まだの場合は)ITC-LMS 内の授業アンケートに回答すること

参考文献

[1] 渡部 有隆，「オンラインジャッジではじめる C/C++ プログラミング入門」，マイナビ，2014 年, https://book.mynavi.jp/ec/products/detail/id=25382

[2] 渡部 有隆，「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」，マイナビ，2015 年, https://book.mynavi.jp/ec/products/detail/id=35408

[3] 秋葉 拓哉, 岩田 陽一, 北川 宜稔,「プログラミングコンテストチャレンジブック」第二版, マイナビ, 2012 年, http://book.mycom.co.jp/book/978-4-8399-4106-2/978-4-8399-4106-2.shtml

[4] Jon Kleinberg and Eva Tardos, 「アルゴリズムデザイン」, (浅野 孝夫，浅野 泰仁，小野 孝男，平田 富夫 訳)，共立出版，2008 年, http://www.kyoritsu-pub.co.jp/bookdetail/9784320122178

[5] T. コルメン，C. ライザーソン，R. リベスト，C. シュタイン,「アルゴリズムイントロダクション」第 3 版 総合版, (浅野 哲夫，岩野 和生，梅尾 博司，山下 雅史，和田 幸一訳)，近代科学社，2013 年, http://www.kindaikagaku.co.jp/information/kd0408.htm

[6] Bjarne Stroustrup, “The C++ Programming Language” (4th Edition), Addison-Wesley Professional, 2013

索引