

問題解決のためのプログラミング一巡り

公開版 (CC BY 4.0) 金子 知適 (kaneko@acm.org)

目次

第 1 章	はじめに	5
1.1	資料の構成	5
1.2	プログラミング言語	5
1.3	教科書・参考書	6
1.4	オンラインジャッジシステム	6
1.5	Aizu Online Judge (AOJ)	6
1.6	入出力と問題の対応	8
1.7	実践上の注意事項	10
第 I 部	入門コース	15
第 2 章	入出力と配列・シミュレーション	16
2.1	最大値, 最小値	16
2.2	計算時間の見積と試行回数	19
2.3	練習問題	23
第 3 章	整列と貪欲法	25
3.1	さまざまな整列	25
3.2	貪欲法	30
第 4 章	動的計画法 (1)	36
4.1	数を数える	36
4.2	最適経路を求めて復元する	41
4.3	さまざまな動的計画法	41
4.4	練習問題	44
第 5 章	分割統治	48
5.1	二分探索	48
5.2	Merge Sort	52
5.3	木のたどり方	54
5.4	空間充填曲線	56

第 6 章	基本データ構造 (string, stack, queue, string, set, map)	59
6.1	文字列 (string) と入出力	59
6.2	スタック (stack) とキュー (queue)	60
6.3	文字列と分割・連結・反転	66
6.4	集合 (set)	67
6.5	連想配列 (map)	69
6.6	練習問題	71
第 7 章	グラフ (1) 全域木	74
7.1	グラフと木	74
7.2	Disjoint Set (Union-Find Tree)	76
7.3	全域木	79
7.4	練習問題	81
7.5	補足: 木に関連する他の話題	84
第 8 章	グラフ (2) グラフ上の探索	90
8.1	グラフの表現: 隣接リストと隣接行列	90
8.2	幅優先探索 (BFS)	92
8.3	深さ優先探索 (DFS)	94
8.4	連結性の判定	96
8.5	二部グラフの判別	98
8.6	様々なグラフの探索	100
第 9 章	グラフ (3) 最短路問題	103
9.1	重み付きグラフと表現	103
9.2	全点对間最短路	103
9.3	単一始点最短路	107
9.4	練習問題	113
第 II 部	トピックス	115
第 10 章	平面の幾何	116
10.1	概要: 点の表現と演算	116
10.2	三角形の符号付き面積の利用	119
10.3	様々な話題	124
10.4	応用問題	126
第 11 章	簡単な構文解析	130
11.1	四則演算の作成	130
11.2	練習問題	136

第 12 章	繰り返し二乗法と行列の冪乗	140
12.1	考え方	140
12.2	言語機能: struct と再帰関数	141
12.3	正方行列の表現と演算	141
12.4	練習: フィボナッチ数	142
12.5	応用問題	144
第 13 章	配列操作	146
13.1	配列, std::vector, std::array と操作	146
13.2	練習問題	149
13.3	いろいろな問題	153
第 14 章	整数と連立方程式	156
14.1	素因数分解、素数、ユークリッドの互除法など	156
14.2	連立方程式を解く	160
14.3	その他の練習問題	162
第 15 章	補間多項式と数値積分	163
15.1	Lagrange 補間多項式	163
15.2	数値積分とシンプソン公式	166
15.3	道具としての Fast Fourier Transform (FFT)	168
第 16 章	区間の和/最大値/最小値と更新	170
16.1	累積和	170
16.2	Binary Indexed Tree (Fenwick Tree)	172
16.3	Segment Tree と Range Minimum Query	175
第 III 部	補遺	178
付録 A	バグとデバッグ	179
A.1	バグの予防とプログラミング作法	179
A.2	デバッグの道具	181
A.3	標本採集: 不具合の原因を突き止めたら	187
付録 B	プログラミング言語と環境の理解	189
B.1	ループ不変条件	189
B.2	再帰	190
B.3	整数型の理解	195
B.4	浮動小数と誤差	197
B.5	構造体 struct	200
付録 C	Ruby	201

C.1	入出力	201
C.2	整列と貪欲法	202
C.3	動的計画法	203
C.4	基本データ構造	204
C.5	グラフ上の探索	207
C.6	最短路	209
C.7	数値積分	209
参考文献		212
索引		213

第 1 章

はじめに

1.1 資料の構成

各章が 90–105 分の演習時間を想定して作られている^{*1}。「例題」やヒントがついた易しい「練習問題」は、主に未経験者を想定して用意されていて、一旦理解した後であれば 5–10 分で解けるものが多い。しかし、初めて取り組む場合は時間を 5 倍程度長めに見積もることをお勧めする。経験者向けには、難易度の異なる複数の練習問題が紹介されている。所要時間は熟達度で異なるが、問題名に印★が一つつくと、難易度が 5 倍程度（たとえば回答作成に要する時間で測ったとして）難化する目安である。また後ろの章の知識を前提としている場合もある。そのため、各章の問題を全て解いてから次に進むのではなく、印なしの易しい問題を解いたら一旦次の章に進むことを勧める。一旦ひと通り例題を解いてどのような話題があるか目を通すと、二週目には解ける問題が増えていることだろう。さらに印★を二つ以上持つ問題は、その章の内容と多少は関係があっても解法と直接の関係がない場合もある。これは、どの戦略が有効かの見極めも、問題解決を学ぶ面白さの一つであるため。

■凡例 資料内へのリンクは深緑で示される (例: 1 章, 文献 [3])。また、外部へのリンクは青で示される (例: <http://www.graco.c.u-tokyo.ac.jp/icpc-challenge/>)。

■教養学部前期課程実践的プログラミング履修 (予定) 者への補足 これから開講されるセミナーが、この資料の予習を前提とすることは*ない*。すなわち、未経験者向けの題材が、経験者向けの練習問題と並んで引き続き提供される。扱うテーマはこの資料と重なる部分もあれば重ならない部分もある。

1.2 プログラミング言語

以下の言語での学習を想定する:

- **C++** (推奨: メインの想定言語である)
- Java
- **Python3** (推奨: ただし一部の問題は実行時間制限で解けない可能性がある)
- Ruby (一部の問題は実行時間制限で解けない可能性がある)

以下の言語での学習は、推奨しない:

^{*1} 当初はそうであったが、整理の都合で現在では分量が適切でない章もあるかもしれない。

- C (理由: 標準ライブラリが少ないため. たとえば連想配列機能)
C 言語使用者は C++ を使うこと. この資料の演習に必要な機能は C++ 全体のほんの一部であるので, そこだけ借りて使いながら, 他は C 言語のつもりで書けば良い. つまり, 一般に C++ を学び直すことよりも苦労は少ないはずである.
- Python2 (理由: 変数のスコープや文字コードなど様々な落とし穴がある)

1.3 教科書・参考書

この資料の読者としては, 繰り返しや条件分岐を短いコードならは思い通りにかける状態であること, 再帰についても習ったことがあることが想定されている. そのため本当に初めてプログラミングに触れる場合は, いったん他著で学ぶことを勧める. 既に購入済みの書籍があればそれで十分だが, 新たに購入する場合は「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」[1] が, AOJ を使っている点で本資料との接続が良い.

本文中で, 参考書 攻略[2] として「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」に, 参考書 [3] として「プログラミングコンテストチャレンジブック第二版」に言及することがある (言うまでもなくこの分野の名著である). また, 学習時間 (と予算) にゆとりがある者には, 「アルゴリズムデザイン」[4] の6章までを時間をかけて読み進めることを勧める. ページ数が多いが, その分丁寧に書かれているので, 類書の中では初学者に適すると思われる (ただし筆者は英語版で読んだので, 日本語版の評価は予想である). さらに深く学ぶ場合は, 「アルゴリズムイントロダクション」[5] も, 時間をかけて学習する価値がある.

1.4 オンラインジャッジシステム

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

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

出典の記述の際に日本国内のACM-ICPC(<https://icpc.iisf.or.jp/>) 及び ACM-ICPC OB/OG 会 (JAG; <http://acm-icpc.aitea.net/>) のものは, 区別がつく範囲で簡略に示した. たとえば国内予選は日本の ACM-ICPC のものを, 模擬国内予選は ACM-ICPC OB/OG 会主催の恒例の練習会を指す.

1.5 Aizu Online Judge (AOJ)

アカウント作成

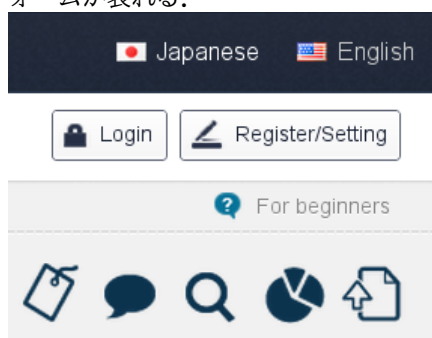
初めての場合はまずAOJのアカウントを作成する. 各システムとも無料で使うことができる. なお, 各オンラインジャッジは, 運営者の好意で公開されているものであるから, 迷惑をかけないように 使うこと. 特

にパスワードを忘れないこと。

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

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



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。Terms of use (https://onlinejudge.u-aizu.ac.jp/#/term_of_use), Judge’s replies (https://onlinejudge.u-aizu.ac.jp/#/judges_replies), チュートリアル (http://judge.u-aizu.ac.jp/onlinejudge/AOJ_tutorial.pdf) などの資料がある。

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

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

1.6 入出力と問題の対応

さっそく，AOJ で1 題回答してみよう．

例題	Rectangle	(AOJ)
たて a cm よこ b cm の長方形の面積と周の長さを求めるプログラムを作成して下さい。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_1_C&lang=jp		

以下の手順で取り組む:

1. 問題を把握する
2. 計算手順を検討する (今回は $a * b$ と $a + b$ を計算すれば良いことを把握する)
3. プログラムを書く. エディタ (Emacs, mi などお好みで) で編集し, ファイルに保存する.
4. 手元のコンピュータで動作確認をする (必須)
5. AOJ に提出して確認する

■解答例 : 以下に各言語の回答例と, $a=3, b=5$ のケースに対する動作確認の方法を示す.

```
Python3
1 a,b = map(int, input().split())
2 area = a*b
3 perimeter = 2*(a+b)
4 print("

```

`input()` で一行読み, `split()` で空白区切りで分割し^{*2}, `map(int, ...)` で分割された各要素を整数に変換している.

上記の内容を `rectangle.py` などと保存したあと, ターミナル 上で実行する. `$` はプロンプトの略記であり, 自身で入力する必要はない (つまり `python3` 以降をタイプする)^{*3}. また, `#` とその右部分は補足説明であり, これも入力の必要はない. 各行の入力終了時に, エンターキーを押すこと. 以下, 斜体はキーボードからの入力を示す.

```
$ python3 rectangle.py
3 5 # キーボードから入力する
15 16 # プログラムの出力の表示
```

```
C++
1 #include <iostream>
2 using namespace std;
```

^{*2} この時, 末尾の改行文字や (もしあれば) 行頭や連続する空白文字も削除される.

^{*3} 詳しくは HWB15.2 を参照 <https://hwb.ecc.u-tokyo.ac.jp/wp/information-2/cui/terminal/>

```

3  int main() {
4      int a,b;
5      cin >> a >> b;
6      // ここで area, perimeterを計算
7      cout << area << ' ' << perimeter << "\textbackslash{}n";
8  }

```

上記のプログラムを、rectangle.cc などに保存する。続いてコンパイルして実行する。
「ターミナル」(MacOSX の場合) の動作例は以下の通り:

```

$ g++ -std=c++11 -Wall -fsanitize=undefined rectangle.cc # コンパイル 1
$ ./a.out # 実行 2
3 5 # キーボードから入力する 3
15 16 # プログラムの出力の表示 4

```

#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 コンパイルオプション である。-Wall は様々な警告を有効にするオプションで、何か警告時に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。-fsanitize=undefined は、実行時エラーを捕捉する機構を有効化する。動作確認の間はつけておくことが望ましい。詳しくは A.2.2 章を参照。



C 言語禁止

本資料を読み進める場合は、C 言語では不十分で、C++ の機能の一部を使う必要がある。必要な部分を少しずつ紹介するので、この時点で iostream, cin, cout などに慣れること。



ブラウザ上のプログラミング禁止

簡単な問題はブラウザ上でコーディングすることもできるかもしれないが、今後扱う複雑な問題はそうではない。手元の PC でコンパイルして、様々なデータで実行できる環境を、この時点で用意しておくこと。

C++

```

1  #include <cstdio>
2  int main() {
3      int a,b;
4      scanf("
5  //ここで area, perimeterを計算
6  printf("
7  }

```

この資料では、C 言語使用者は C++ に移行するよう推奨するが、C++ を用いる場合でも入出力は C の scanf, printf を用いて良い。

Ruby

```
1 a,b = gets.split("_").map(&:to_i)
2 area = a*b
3 perimeter = 2*(a+b)
4 print sprintf("

```

上記の内容を `rectangle.rb` などと保存したあと、ターミナル上で実行する

```
$ ruby rectangle.rb 1
3 5 # キーボードから入力する 2
15 16 # プログラムの出力 3
```

Java の場合、オンラインジャッジシステムの制限で、常に `Main` というクラスに回答を書く必要がある。そのためには `Main.java` というファイル名で保存する必要がある。問題毎にフォルダを作成し、そこで作業すること。

Java

```
1 import java.util.Scanner;
2 public class Main {
3     public static void main(String[] args) {
4         Scanner scanner = new Scanner(System.in);
5         int a = scanner.nextInt(), b = scanner.nextInt();
6         int area = a*b;
7         int perimeter = 2*(a+b);
8         System.out.println(area+"_"+perimeter);
9     }
10 }
```

```
$ javac Main.java 1
$ java Main 2
3 5 # キーボードから入力する 3
15 30 # プログラムの出力の表示 4
```

1.7 実践上の注意事項

1.7.1 フォルダの保存

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

「ターミナル」(MacOSX の場合) の動作例は以下の通り:

```

$ mkdir programming 1
# (フォルダ作成, 最初の一回のみ) 2
$ mkdir programming/chapter1 3
# (各章毎に行う) 4
$ cd programming/chapter1 5
# (カレントディレクトリを変更. $HOME/programming/chapter1/ 以下にソースコードを保存) 6

```

1.7.2 標準入出力とリダイレクションによるファイルを用いたテスト

この節ではデータがあるだけ読み込む方法とファイルを用いたテストの方法を紹介する。この節の内容は、今ただちに身につける必要はないが、第2,3章に取り組む過程で身につけておくことが望ましい。

巨大な入出力を扱う場合は、キーボードから手で入力することは適切でない。(手で入力すると、タイプミスのリスクがある。コピーペーストすると多少改善するが、データ量に上限があり、また範囲選択のミスの余地が依然として残る。プログラムの正しさを検証する際には、他の不確定要因は100%取り除き、プログラムそのものに集中することが望ましい。)

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

例題

年を読み込んで、うるう年かどうかを判定し、日数を出力するプログラムを作る

(いい加減な) プログラム例 (C++): leap.cc

```

C++ 1 // 4年に一度?
2 #include <iostream>
3 using namespace std;
4 int main() {
5     int year;
6     while (cin >> year) { // cinはboolに自動変換されるので入力が読める限り
    ループ
7         if (year
8             cout << 366 << endl;
9         else
10            cout << 365 << endl;
11        }
12    }

```

■コンパイル (前述の通り\$記号は、ターミナルへのコマンド入力を示す)

C++ の場合:

```

$ g++ -Wall leap.cc 1

```

Java の場合:

```
$ javac Main.java 1
```

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

```
$ ./a.out 1
2004 2
366 3
1999 4
365 5
1900 6
366 7
2000 8
366 9
^D 10
```

■ファイルを用いたテスト 実行するたびに毎回キーボードをタイプするのは煩雑であるから、自動化したい。そこで、リダイレクションとファイルを用いたテストを解説する。早い段階で身につけることが望ましい。

正しい入出力例をエディタで作成し、cat コマンドで中身を確認する:

```
$ cat years.input 1
2004 2
1999 3
1900 4
2000 5
$ cat years.output 6
366 7
365 8
365 9
366 10
```

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

```
$ ./a.out < years.input 1
366 2
365 3
366 4
366 5
```

実行結果をファイルに保存 (画面に表示する代わりにファイルに書き込む):

```
$ ./a.out < years.input > test-output 1
$ cat test-output 2
366 3
365 4
366 5
366 6
```

`diff` を用いた自動的な比較:

```
$ diff -u test-output years.output 1
--- years.output      Fri Oct 14 10:53:52 2005 2
+++ test-output Fri Oct 14 10:53:56 2005 3
@@ -1,4 +1,4 @@ 4
  366 5
  365 6
-365 7
+366 8
  366 9
```

4 行目あたりが違うことを教えてくれる


資料:

- HWB 15 コマンド
<http://hwb.ecc.u-tokyo.ac.jp/wp/information-2/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/wp/information-2/filesystem/cui-fs/>

1.7.3 ジャッジデータのダウンロードと手元での実行

プログラムを提出した際に、Accepted ではなく Wrong Answer や Time Limit Exceeded, Rutime Error などとなる場合がある。

仮に今 ITP1_4_D という問題を解いていて、Wrong Answer になったとする。

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などに保存されたとする。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 1
28 999997 724134219 2
```

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。

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 1
$ cat my-output.txt 2
1 17 37 3
```

2つのファイル比較:

```
$ diff -u sample-output.txt my-output.txt 1
```

(差分がある場合のみ、出力がある)

第Ⅰ部

入門コース

第 2 章

入出力と配列・シミュレーション

概要

様々な解法につながる基本として、入力データを逐一調べる問題から始めて、配列と周辺の基本操作を取りあげる。また、プログラムを一度に作成するのではなく、部品ごとに作ってテストするスタイルを身につける。さらに、データの量に応じて、適切なアルゴリズムが必要になることを経験する。

2.1 最大値, 最小値

例題

ICPC Score Totalizer Software

(国内予選 2007)

入力として与えられる数値列から最大値と最小値を除いた平均値を求めよ。

入力はそれぞれが競技者の演技ひとつに対応するいくつかのデータセットからなる。入力のデータセット数は 20 以下である。データセットの最初の行はある演技の採点に当たった審判の数 n ($3 \leq n \leq 100$) である。引き続く n 行には各審判のつけた 点数 s ($0 \leq s \leq 1000$) がひとつずつ入っている。 n も各 s も整数である。入力中にはこれらの数を表すための数字以外の文字はない。審判名は秘匿されている。入力の終わりはゼロひとつの行で示される。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1147&lang=jp>

“Sample Input” を順に解釈すると、以下のように 4 つのデータセットからなることが分かる。

- (N=) 3 (S=) 1000 342 0
- (N=) 5 (S=) 2 2 9 11 932
- (N=) 5 (S=) 300 1000 0 200 400
- (N=) 8 (S=) 353 242 402 274 283 132 402 523
- 0 (入力の終わり)

#-----

3 # データは 3 つ, N=3

```
1000 # 最大値
342
0 # 最小値
#-----
5 # データは 5 つ, N=5
2 # 最小値
2
9
11
932 # 最大値
#-----
5 # データは 5 つ
300
1000 # 最大値
0 # 最小値
200
400
#-----
8 # データは 8 つ
353
242
402
274
283
132
402
523
#-----
0 # これでおしまい
```

同様に“Sample Ouput”を解釈すると、上記の各データセットに対して一つ数値が出力されていることが分かる。

- 342 (そのまま)
- 7 (2, 9, 11 の平均)
- 300 (300, 200, 400 の平均)
- 326 (... の平均)

2.1.1 プログラム作成

以下に、審判の得点の合計を計算するプログラムと最大得点を計算するプログラムを示す。(問題が求めるものは合計や最大値ではないので、これらは問題への直接の回答ではない)。必要ならばこれらのプログラムを参考に、問題への回答を作成せよ。

C++

```

1  #include <iostream>
2  using namespace std;
3  int N, S;
4  int main() {
5      while (cin >> N && N>0) {
6          int sum = 0;
7          for (int i=0; i<N; ++i) {
8              cin >> S;
9              sum += S; // std::accumulate を使っても良い
10         }
11         cout << sum << endl;
12     }
13 }
```

C++

```

1  // (一部略)
2      while (cin >> N && N>0) {
3          int largest = 0;
4          for (int i=0; i<N; ++i) { // std::max_element を使っても良い
5              cin >> S;
6              if (largest < S) largest = S;
7          }
8          cout << largest << endl;
9      }
```

C 言語から C++ に移行する場合は、(基本は共通なので) さしあたり 網掛け 部分を覚えて使えるようになると良い。コンパイルの仕方などは 1.7 節を参照。



C 言語禁止

本資料を読み進める場合は、C 言語では不十分で、C++ の機能の一部を使う必要がある。必要な部分を少しずつ紹介するので、この時点で `iostream`, `cin`, `cout` などに慣れること。



関数を使おう

`max`, `min`, `sum` などは関数を使っても良い。

Python3

```

1  while True:
2      N = int(input())
3      if N == 0:
```

```

4         break
5     S = []
6     for i in range(N):
7         S.append(int(input()))
8     print(sum(S))
9     print(max(S))

```

なお、Python3 で整数除算を行うには/に代えて//という演算子を用いる。

2.1.2 動作テスト

サンプル入力を用いたテスト

問題文中の“Sample Input”に対する出力が合うことを確認してみよう。

審判データを用いたテスト

この問題は審判が用いた秘密の入出力が公開されている。これを利用して、プログラムの誤りの有無をさらに手元で試験することができる。(プログラムが“Sample Input”には正しく振舞うが、他のデータには誤ることを発見できるかもしれない)

- 入力 <http://www.logos.ic.i.u-tokyo.ac.jp/icpc2007/jp/domestic/datasets/A/A1>
- 出力 <http://www.logos.ic.i.u-tokyo.ac.jp/icpc2007/jp/domestic/datasets/A/A1.ans>

出力の一致を確認するには、diff コマンドを用いる。実行方法などは 1.7 節を参照。

```

$ python3 icpc.py < A1 > my-out.txt      1
$ diff -u my-out.txt A1.ans              2

```

```

$ ./a.out < A1 > my-out.txt              1
$ diff -u my-out.txt A1.ans              2

```



後回し禁止

この時点で (次の章に進む前に) リダイレクションと diff の操作を覚えること。

2.2 計算時間の見積と試行回数

コンピュータが得意な解法は全部を力づくで試すことである。実際に多くの問題をそれで解くことが出来る。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。

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

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1192&lang=jp>

(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、C++ なら今回は必須ではない。Python の場合は、多少の工夫が必要である。下記のコードでは、ある組 (a_0, b_0) の税込価格が S を越えたら、 $b > b_0$ での組 (a_0, b) の税込価格も S を必ず超えるので、検討不要であるという性質を利用した。



関数必須 (横着禁止)

必ず関数 `tax` を作成すること。徐々に複雑なプログラムを扱う際に、関数の作成能力は重要となる。この時点で身につける。

Python3

```
1  def tax(p, x):
2      return p*(100+x) // 100      # 整数除算
3  def solve(X, Y, S):
4      for a in range(1, S):
5          for b in range(1, S):
6              sum = tax(a, X) + tax(b, X)
7              if sum == S:
8                  # 新税での tax(a, Y) + tax(b, Y) について検討
9              if sum > S:
10                 break          # b 増加ならば sum 増加のため
11  return best
```

```

12 while True:
13     X,Y,S = map(int, input().strip().split('_'))
14     if X == 0:
15         break
16     print(solve(X,Y,S))

```



TLE (Time limit exceeded)

Python の場合は C++ よりも実行速度が遅くなるため、C++ と同じ方針では AC を得られない場合がある。例えば、上記のサンプルコードでは 9,10 行目に高速化のための工夫を施している。



税込み v.s. 税抜き

本資料では、税抜価格と税率から税込価格を求める関数を作成した。向きを反対にして、税込み価格から税抜価格を求めるアプローチはどうだろうか。実はそちら向きは罠が多数あるのでお勧めしない(興味がある場合はお勧めの方法で AC 後に理由を考えてみよう)。

■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++

```

1 #include <iostream>
2 using namespace std;
3 int X, Y, S;
4 int solve() {
5     ... // X,Y,Sについて、最大値を計算
6 }
7 int main() {
8     while (cin >> X >> Y >> S && X>0) {
9         cout << solve() << endl;
10    }
11 }

```

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.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 1
$ diff -u A1.ans my-output.txt 2
```

差があれば行数が表示される。

■計算量の簡単な検討 (estimation of the number of trials) 多くのコンテストの問題では実行時間に制限がついているので、実行時間を見積もり間に合う解法を考案することが必要な場合もある。またコンテストを離れて実用に用いるプログラムでも、何らかの実行速度に関する要請がある場合が多い。プログラムを書き終えてから速度に問題があることが分かると、書き直しが困難な場合もあるので、プログラムを書く*前*に何らかの見通しを得ておくことが望ましい。

現実の計算機は複雑な装置であるから、正確な実行時間の予想は簡単ではない。よって単純なモデルに基づく目安を考える。たとえば、プログラムを実行する過程で、加減乗除のような基本演算が何回行われるかを数える。このモデルでは加算と除算では速度が異なりうるとか、同時に二つの命令を実行される場合があるなどの、細かい点は無視している。目安であるので現実との対応関係は別に議論が必要だが、役に立つ場面も多い。

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、 $f(S) \leq S^2 \leq 1000^2 = 10^6$ である。

表 2.1 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 2.1 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。^{*1}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。

本資料の範囲では、この表を信じて問題ない場合がほとんどであるが、正確な予測のためには、キャリブレーションを行う。すなわち、作成した解法が入力に対応する数値 N に関しておよそ何回の基本演算を行うかを見積もったら、いくつかの N に対応する入力で実験し、実際の計算秒数との対応を取る。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。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。問題ごとの秒数制限と、オンラインジャッジのハードウェアを考慮して検討する。



Python の場合は C++ よりも実行速度が遅くなる

問題によるが20-200倍余裕をもって見積もると良い。オンラインジャッジで時間制限はPythonに適切とは限らないので、ジャッジデータをダウンロードして正答を確認できればそれでも良い。

^{*1} 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。

問題

Space Coconut Grab

(模擬国内予選 2007)

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

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

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

変数 x, y, z の動く範囲を考えると、それぞれ $x: [0, E]$, $y: [0, \sqrt{E}]$, $z: [0, \sqrt[3]{E}]$ である。それら (x, y, z) の組み合わせは、 E の最大値は $1,000,000 = 10^6$ であるから、最大 $10^6 \cdot 10^3 \cdot 10^2 = 10^{11}$ である。制限時間が 8 秒であることを加味しても、この方針では間に合いそうにない。

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

Python-TLE

残念ながら上記の方針では Python では時間制限を超過 (TLE) する。対策としては、2 重ループを避けて z に関する 1 重ループのみにすれば良い。単調性から y に関するループを省くことが出来る。

```
Python3 1 import math
        2 # ... (中略) ...
        3 y = int(math.floor(math.sqrt(E - z**3) + 1e-7))
```

2.3 練習問題

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&lang=jp>

注: 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間に合わないので、工夫した方法が必要である。正方形を一つずつ数えると数が多すぎるので、一定の性質を持つ正方形をまとめて数えるような方法が必要となる。

ヒント: 後の章で扱う標準データ構造を活用する方法と, 正方形の初等的な性質を利用する方法がある (について考えてみよう).

第3章

整列と貪欲法

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、まずはその使い方を習得しよう。この資料では整列された結果を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、たとえば参考書 攻略 [2, pp. 51–(3 章)] 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めの問題も含まれる。

3.1 さまざまな整列

3.1.1 数値の整列

C++ と Ruby では、標準関数として `sort` や `sort!` が用意されている。

```
C++
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4 int A[5] = {3, 5, 1, 2, 4};
5 int main() {
6     sort(A, A+5); // 半開区間 [1, r) で指定する。sort(&A[0], &A[5]) と同じ意味
7     ... // cout に A を出力してみよう
8 }
```

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。配列でなく `vector` や `array` の範囲を指定するには、`A.begin()`、`A.end()` という表記を用いる。この `begin` や `end` はポインタを一般化したイテレータを返す関数で、名前の通りの場所を指し示す。他に `A.begin()+2`、`A.begin()+5` のように一部の区間を指定することもできる。さらに C++11 以降では `begin(A)` とすることで配列も `vector` も共通に扱うことができる。

Python3

```

1 a = [3, 5, 1, 2, 4]
2 a.sort()
3 a
4 # [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&lang=jp>

回答例:

C++

```

1 int N, A[1000000+10];
2 int main() {
3     cin >> N;
4     for (int i=0; i<N; ++i) cin >> A[i]; // Aの入力
5     ... // Aをソートする
6     for (int i=0; i<N; ++i) cout << (i?" ":"") << A[i]; // Aの出力
7     cout << endl;
8 }

```

出力部分の `(i?" ":"")` は、要素の間にのみ空白文字を入れるための微調整 (先頭 `i==0` には空白を入れない)。

3.1.2 文字列の整列

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&lang=jp>

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++

```

1 #include <algorithm> // sort のため
2 #include <string> // string (文字列) のため
3 #include <iostream>
4 using namespace std;

```

```

5  string A[1000];
6  int N;
7  int main() {
8      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 }

```

3.1.3 ペアと整列

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。

```

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 }

```

Python や ruby の場合は列 (配列やリスト) を手軽に使えるので, (取り立てて pair を区別せず) 列を使う。

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++ 1  #include <utility> // pair のため
    2  #include <algorithm> // sort のため

```

```

3  #include <iostream>
4  using namespace std;
5  int main() {
6      pair<int,int> a[3]; // 整数のペア
7      a[0] = make_pair(170,60);
8      a[1] = make_pair(180,90);
9      a[2] = make_pair(170,65);
10
11     for (int i=0; i<3; ++i) // a[0] から a[2] まで表示
12         cout << a[i].first << ' ' << a[i].second << endl;
13     // 170 60
14     // 180 90
15     // 170 65 と表示されるはず
16
17     sort(a, a+3); // a[0] から a[2] まで整列
18
19     for (int i=0; i<3; ++i) // a[0] から a[2] まで表示
20         cout << a[i].first << ' ' << a[i].second << endl;
21     // 170 60
22     // 170 65
23     // 180 90 と表示されるはず
24 }

```

Python3

```

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

```

3つ以上の組

C++11 では、3 つ以上の組を表す tuple 型が用意されている。^{*1} それより前の C++ では pair<int,pair<int,int> >などと pair を重ねて用いていたところが、簡潔に書けるようになった。

C++11

```

1  #include <tuple>
2  #include <iostream>
3  #include <algorithm>
4  using namespace std;
5  int main() {
6      tuple<int,int,int> a = {3,1,4};
7      cout << get<0>(a) << "\n";    // 3
8      cout << get<1>(a) << "\n";    // 1
9      cout << get<2>(a) << "\n";    // 4
10
11     tuple<int,int,int> array[] = {{3,1,4}, {1,5,9}, {2,6,5}};

```

^{*1} <http://en.cppreference.com/w/cpp/utility/tuple>

```

12  sort(array, array+3);
13
14  tuple<int,int,int> t = array[0];
15  cout << get<0>(t) << "\n";    // 1
16  cout << get<1>(t) << "\n";    // 5
17  cout << get<2>(t) << "\n";    // 9
18  }

```

3.1.4 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++ 1  int A[5] = {3,5,1,2,4};
    2  int main() {
    3      sort(A,A+5);
    4      // C++11 なら sort(begin(A),end(A));
    5      reverse(A,A+5); // 与えられた範囲を逆順に並び替え
    6      ... // cout に A を出力してみよう
    7  }

```

```

Python3 1  a = [3,5,1,2,4]
    2  a.sort()
    3  a.reverse() # a を逆順に並び替え

```

2. `reverse_iterator` を使う

```

C++14 1  sort(rbegin(A), rend(A));

```

```

Python3 1  a.sort(reverse=True)

```

3. 比較関数を渡す (汎用的)

```

C++11 1  int A[5] = {3,5,1,2,4};
    2  int main() {
    3      sort(begin(A),end(A), [](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)

```

3.2 貪欲法

3.2.1 良い順に使う

問題	カントリーロード	(UTPC 2008)
<p>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ない、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。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2104&lang=jp</p>		

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 }
```

Python3 の入力例: 与えられた 1 行を配列として読み込む場合は `list(...)` で変換する

Python3

```

1  N, K = map(int, input().strip().split(' '))
2  X = list(map(int, input().strip().split(' '))) # リストへの変換を明示
```

☠ 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ず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.

問題

Stripies

(Northeastern Europe 2001)

質量が m_1, m_2 である 2 体合体すると, $2\sqrt{m_1 \cdot m_2}$ となる種族がある. 初期状態を与えられるので, 全てが合体した時の最小の質量を求めよ.

<http://poj.org/problem?id=1862>

小数点以下 3 桁を出力するには `printf("%.3f\n", ret);` を用いる.

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 $\langle D, P \rangle$ が N 個.

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

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, 危険なままの区間に道が分かれる. 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, $\langle \text{危険度}, \text{長さ} \rangle$ のペアで道を表現し, 危険な順に整列しておくとう便利である. (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)

回答例 (入力と整列):

C++

```
1 int N, M;
2 pair<int,int> PD[10010];
3 int main() {
4     while (cin >> N >> M && N) {
5         int d, p;
6         for (int i=0; i<N; ++i) {
7             cin >> d >> p;
8             PD[i] = make_pair(p, d);
9             // PD[i].first は道 i の危険度
```



```

10      // PD[i].second は道 i の長さ
11    }
12    ... // PD を大きい順に整列しよう
13    // 整列がうまく行ったか, PD を表示してみよう
14    // うまく整列できたら, 次は答えを計算しよう
15  }
16 }

```

回答例 (答えの計算):

```

C++ 1  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; // 雇う区間
    8    S -= 道[i]の危険度 * guarded;
    9    M -= guarded;
   10  }
   11  S が答え

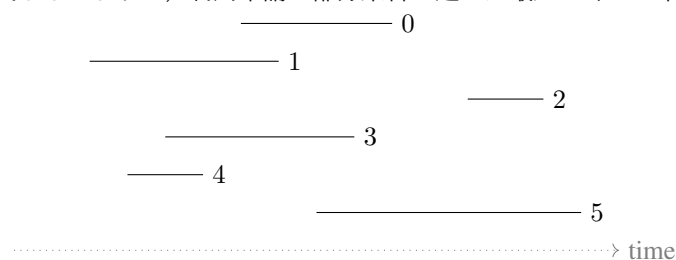
```

- 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.

3.2.2 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)

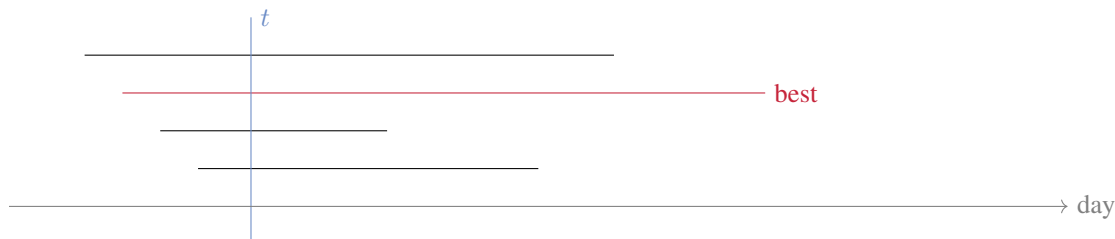
■正しいアルゴリズム

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

■区間を扱う練習問題 以下の問題は, 区間スケジューリング問題とは異なるが, 区間を扱う問題である.

問題	Cleaning Shifts	(USACO 2004 December Silver)
<p>連続する T 日の掃除当番を決めたい。各「牛」は何日目から何日目まで (境界を含む) 区間働けることがわかっている。当番が不在の日の内容に牛を配置するとき, 最低何頭必要か答えよ。不可能な場合は -1 を出力せよ。</p> <p>http://poj.org/problem?id=2376</p>		

ヒント: 当番に穴を空けない中で, もっとも遅くまで担当できる牛を採用してゆく。



回答例 (入出力)

```

C++
1  int  /*牛の総数*/N, /*掃除日数*/T;
2  pair<int,int> C[25010]; // C[i].first が担当開始日, C[i].second が担当終了日
3  int solve() {
4      // N, T, C を元に回答を計算する
5  }
6  int main() {
7      scanf("
8      for(int i=0; i<N; i++)
9          scanf("
10         printf("
11     }

```

回答例 (計算)

```

C++
1  int solve() {
2      sort(C, C+N); // 掃除開始日の早い順に整列
3      int /*牛番号*/i = 0, /*次の担当の掃除開始日*/t = 1, /*採用数*/c = 0;

```

```

4      while (i < N && t <= T) {
5          int best = 0; // 次の雇う予定の牛の担当終了日
6          while (i < N && 牛 i の担当開始が t より遅くならない間) {
7              ... // 牛 i の担当終了が best より遅ければ best を更新
8              ++i;
9          }
10         if (担当可能な牛が一頭も居なければ (best < t)) return -1;
11         t = best + 1;
12         ++c;
13     }
14     return /* T まで掃除終了しているか ? */ ? c : -1;
15 }

```

戦略: $t-1$ 日目まで掃除が終わっていたとする。 t 日目を含む区間に掃除をできる牛がいなければ、解はない (-1 を出力)。もし複数の牛が t 日目を担当可能であれば、なるべく長く担当してもらえる牛を雇うのが良い (*). その牛が s 日まで担当したとすると、 $t = s + 1$ として、全体の区間が終わるまで、牛を雇い続ける。

他の牛を雇っても最適解になる可能性はあるが、(*) の戦略をとった場合と比べて、雇う牛の数を減らすことはできないことが証明できる。

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

注意: $y=0$ の島が存在する模様

<http://acm.pku.edu.cn/JudgeOnline/problem?id=1328>

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
- 左から順に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証明せよ

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)。

3.2.3 様々な問題

問題	Make Purse Light	(模擬国内予選 2005)
<p>財布の中身を軽くする</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2007&lang=jp</p>		
問題	Fox and Card Game	(Codeforces 388 C)
<p>先手は山の一番上からカードをとり、後手は一番下から取る時、それぞれが最善を尽くした時の得点を求める。</p> <p>http://codeforces.com/problemset/problem/388/C</p>		
問題	Shopping	(アジア大会 2014)
<p>直線上に配置された複数の店を制約を満たしながら買い物をして左から右に抜ける時の、必要な移動の最小を求める。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1347&lang=jp</p>		
問題	Dinner★	(夏合宿 2014)
<p>食堂の食事と自炊を組み合わせる N 日間の幸福度を最大化する。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2642&lang=jp</p>		
問題	Ploughing★★	(13th Polish Olympiad in Informatics)
<p>畑を、縦 (幅 1 列) か横 (1 行) にスパッと切りとることを繰り返して、分けする。どの範囲も数の合計が K 以下になるようにする。条件を満たす最小の分割数を求めよ。</p> <p>https://szkopul.edu.pl/problemset/problem/6YiP6JA5U15hY94pLwuHoYPg/site/</p>		

第 4 章

動的計画法 (1)

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も、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。



フィボナッチ聖人の彫像

(Camposanto, Pisa)

4.1 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、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):

$$F_i = \begin{cases} 1 & i = 0 \\ 1 & i = 1 \\ F_{i-1} + F_{i-2} & (\text{otherwise}) \end{cases} \quad (4.1)$$

上記の定義をそのまま再帰で定義すると以下ようになるだろう:

Python3

```
1 def fib(n):
2     print("fib", n) # 関数呼び出しの際に引数を表示 (動作確認後に消す)
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。n の増加に伴い急速に遅くなる。一例として python の `timeit` モジュールを使った実行時間測定例を示す。

```

Python3 1 import timeit
        2 print(timeit.timeit("fib(10)", globals=globals(), number=10))
        3 print(timeit.timeit("fib(20)", globals=globals(), number=10))
        4 print(timeit.timeit("fib(30)", globals=globals(), number=10))

```

```

$ python3 fib.py
0.0001415439764969051
0.01775405099033378
2.1516372300102375

```

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 (4.1) 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 }

```

この章で扱う問題は、求めたい問題の答え (e.g., F_{100}) が部分問題の答え (F_n , $n < 100$) から効率的に計算可能なものを取り扱う。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、一見回り道のようにも 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、メモ化や繰り返し自乗法による $O(\log N)$ の計算方法については^{*1}、12 章を参照されたい。

この節の目標は、部分問題の解の関係を表す、式 (4.1) のような漸化式から、解を求め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数を, 適当に整形して出力する.

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

各段への登り方の数を配列で表現する. 漸化式: 部分問題として, 下から i 段目に到達する登り方を A_i 通りとする. 登る前は $A_0 = 1$, 最終的に知りたいのは A_n である.

$$A_i = \begin{cases} 1 & i = 0 \\ A_{i-1} & i = 1 \\ A_{i-1} + A_{i-2} & i = 2 \\ A_{i-1} + A_{i-2} + A_{i-3} & (\text{otherwise}) \end{cases} \quad (4.2)$$

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.

回答例 (数の計算)

C++

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4 int A[128], N;
5 int main() {
6     A[0] = 1;
7     for (int i=1; i<=32; ++i) {
8         A[i] = A[i-1];
9         if (...) A[i] += A[i-2];
10        if (...) A[i] += A[i-3];
11    }
12    // A[.] を適当に出力してみる
13 }
```

Python3

```
1 A = [0 for _ in range(32)]
2 A[0] = 1
3 for i in range(1, 32):
4     A[i] = A[i-1]
5     if i > 1:
6         A[i] += A[i-2]
7     if i > 2:
8         A[i] += A[i-3]
```

💥 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、 $i \leq 1$ 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;
```

整数で切り上げるには、割る前に除数-1を足せば良い。Python3では整数除算には`//`を用いる。浮動小数と`ceil`を用いると計算誤差が発生するので注意。

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

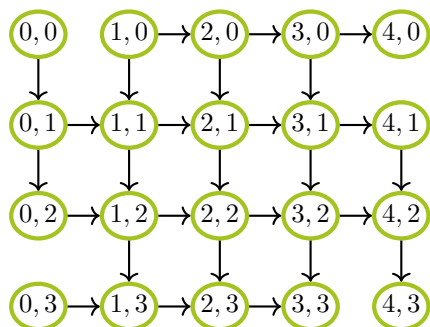
(UTPC2009)

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、マタタビの落ちている道は通れない。

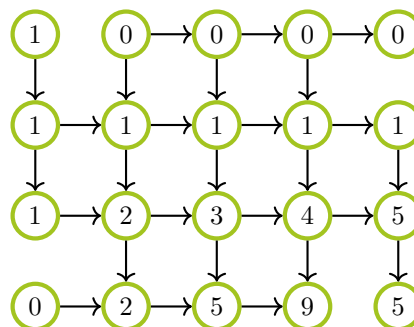
<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2186&lang=jp>

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いくのが自然な解法で、サンプル入力3つめの状況は、下の図のようになる。

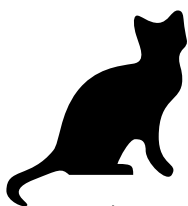


交差点の座標と通れる道 (右または下に移動可)



各交差点に到達可能な経路の数

部分問題として、ある交差点に (x, y) に到達可能な経路数を考え、 $T_{x,y}$ と表記する。この値はそこに一步で到達できる交差点 (通常は左と上) 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。



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

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

- 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、 $\text{Vert}[x][y]$ と $\text{Horiz}[x][y]$ 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 $\text{Vert}[x][y] == \text{true}$ 、そうでなければ $\text{Vert}[x][y] == \text{false}$ とする。同様に、左から移動可能かで $\text{Horiz}[x][y]$ 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.

4.2 最適経路を求めて復元する

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.

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

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



4.3 さまざまな動的計画法

■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. 品物の種類と重さの取りうる数値の積が一定以内なら動的計画法で解くことが出来る. なお, 液体のように自由に量を調整できる場合は-全く別の問題であり-, 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 (動的計画法は不要である).

問題

Combinatorial - 0-1 Knapsack Problem

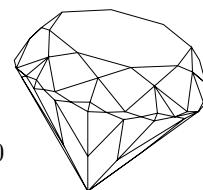
(AOJ)

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。

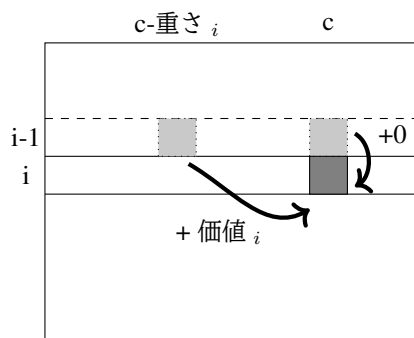
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_B&lang=jp

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、ナップザックの重さ制限が 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。参考書 攻略[2, pp. 416–] 参照。



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ザック問題は、1個限定のナップザック問題と同じ計算量のオーダー $O(NW)$ 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ザ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).

■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 とする. 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&lang=jp		

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. 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.

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

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. 情報科学の「パターン認識」の章や参考書 攻略 [2, pp. 253–] を参照.

補足: この問題は, Python では制限時間に収めることが難しい. TLE の場合は, データをダウンロードして手元で答えを検証すると良い.

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		

■最長増加部分列 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 (5.1 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. 問題末尾の解説や参考書 攻略 [2, pp. 421–] 参照.

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		

4.4 練習問題

問題	Minimal Backgammon	(アジア地区予選 2007)
<p>直線上のすごろくでゴールできる確率を求めよ.</p> <p><code>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1277&lang=jp</code></p>		

問題	Coin Changing Problem	(AOJ)
<p>ぴったり支払うときの、コインの最小枚数を求めよ.</p> <p><code>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_A&lang=jp</code></p>		

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. 参考書^{攻略}[2, pp. 412–] 参照.

問題	Eleven Lover*	(模擬地区予選 2009)
<p>数字列中の 11 の倍数を数える</p> <p><code>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2182&lang=jp</code></p>		

方針例: 左から

計算する.

問題	Restore Calculation*	(アジア地区予選 2013)
<p>?を含む, 二つの整数とその加算結果が与えられる. 演算の整合性を満たすような?の埋め方が 何通りあるか求めよ.</p> <p><code>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2566&lang=jp</code></p>		

問題	Magic Slayer★	(夏合宿 2009)
<p>単体攻撃の魔法と全体攻撃の魔法を上手に使って、敵を全滅させろ。各魔法には、単体/全体の区別の他に、使用 MP と与えるダメージ (Damage) が定められている。敵はいくつのダメージで倒されるかとして HP の数値が与えられる。</p> <p><code>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2156&lang=jp</code></p>		

解説: `http://acm-icpc.aitea.net/index.php?plugin=attach&refer=2009%2FPractice%2F%B2%C6%B9%E7%BD%C9%2F%B9%D6%C9%BE&openfile=2b.pdf`

魔法の名前は無視して良い。

単体魔法のみで敵が一体の場合を考える。合計でダメージ d を与えるのに必要な MP の最小値を A_d とすると

$$A_d = \begin{cases} 0 & d \leq 0 \\ \min_k (A_{d-\text{Damage}_k} + \text{MP}_k) & d > 0 \text{ 最後に魔法 } k \text{ を使った} \end{cases} \quad (4.3)$$

敵 N 体を単体魔法のみで倒す場合は、それぞれの HP に必要な分を合計すれば良い。

全体魔法がある場合は、全体に合計ダメージ d を与えるのに必要な最小な MP を同様に計算し、組み合わせる。

問題	Mushrooms★	(Algorithmic Engagements 2010)
<p>直線を歩いてキノコを採集する。</p> <p><code>https://szkopul.edu.pl/problemset/problem/KVIxa_im2wGYJX99NB31p_nC/site/</code></p>		

scanf 推奨。

問題	輪番停電計画★	(国内予選 2011)
<p>上手に区間を分割する。</p> <p><code>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1176&lang=jp</code></p>		

長方形に対する値を管理するタイプ。

問題	Quest of Merchant★	(夏合宿 2011)
<p>様々な行動を総合して (問題文参照) 収益を最大化せよ。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2296&lang=jp</p>		

問題	My friends are small	(夏合宿 2010)
<p>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</p>		

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-8elttUgorfbix9t0lfvYN3g3/view>

問題	Hakone★★	(夏合宿 2012)
<p>(箱根駅伝の) 順位変動 (このチームは順位をあげて 5 位で通過などの情報) から, 前の中継所での順位として何通り考えられるか求める。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2439&lang=jp</p>		

問題	Barricades	(Algorithmic Engagements 2007)
<p>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。</p> <p>https://szkopul.edu.pl/problemset/problem/1sX3vqLjiqkpxBNI-sh8UC2m/site/</p>		

問題	Army Training★★	(Algorithmic Engagements 2010)
<p>平面上の 1000 個までの点を与えられる. それらを適当につないで単純多角形を作った際に, 領域に含まれる点の個数を数えよ.</p>		

(問題 続き)

https://szkopul.edu.pl/problemset/problem/nXF1qOIv5S88utFPI2V_0gt3/site/

第 5 章

分割統治

概要

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

この章では再帰 (recursion) を取り扱う。再帰に慣れていない場合は、付録 B.2 章を適宜参照のこと。

この章では、アイ (i) とジェイ (j)、イチ (1) とエル (l) が特に紛らわしいので注意のこと。

5.1 二分探索

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 ページと一ページず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。

なお、次の節で紹介するマージソートや inversion count では、分割後に両方の領域について処理する必要がある。

5.1.1 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 sort(S, S+L); // 準備: 事前に配列 S 内を昇順に並び替えておく
```

```

5  ...
6      if (binary_search(S, S+L, a)) {
7          // aがS内にある
8      }

```

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. $l+n \leq r$ 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 ($A[m] \leq value$ 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      // 前提:
3      // - array は昇順に整列されている,
4      // - 半开区間 [left, right) は有効な範囲
5      // 可能性
6      // A. array[left] ≤ value ≤ array[right-1] (答えがありうる)
7      // B. value ≤ array[left] (false 確定)
8      // C. array[right-1] ≤ value (false 確定)
9      while (left + 1 < right) {
10         // ループ不変条件: ループ開始時に成立した A, B, C のどれかが終了時も成立
11         int med = (left+right)/2;
12         if (array[med] > value) right = med;
13         else left = med;
14     }
15     return left < right && array[left] == value;
16 }

```

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.

複雑なループを含むプログラムの正しさについて考える道具として, ループ不変条件 という概念がある. 必要に応じて, 付録 B.1 章も参照のこと.

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. S と T の要素数はそれぞれ 100 以内. 配

(例題 続き)

列に含まれる要素は, 0 以上 10^7 以下.

(注意) T の要素は互いに異なるが, S の要素は重複がある場合がある.

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

🚨 問題の仕様に注意

「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」という意味は, $S=[1, 1, 1]$, $T=1$ とすると答えは 1 ということである.

5.1.2 制約を満たす最小値を求める

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積む切するためには, 各トラックの最大積載量は最低どれだけ必要か. 「トラックの最大積載量」の最小値を求めよ.

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

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. 求める値「積みきれない最小値」は (積載量が整数なので), 「積みきれない最小値」= 「積みきれない最大値」+1 という関係から, 「積みきれない最大値」を求める. 「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         // loop 不変条件: l では積めない, h で積める
9         int m = (l+h)/2;
10        if (ok(m)) h = m; else l = m;
11    }
12    printf("

```

```
13 }
```

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

問題	Aggressive Cows*	(USACO 2005 February Gold)
直線上に N 棟の牛舎 (個室) がある. C 頭の牛を, なるべく互いを離して入れたい. (cin の代わりに scanf を使ってください) http://poj.org/problem?id=2456		

今回は条件を満たす最大値がほしいので, 下限 l を増やせる条件を関数 `ok` として表現する. 求める答えは l となる. あらかじめ小屋の候補を昇順に並べておくと, `ok` 関数は,

- 牛を左端の小屋に配置する (小屋がなければ失敗)
- 距離 M 以内の小屋を考慮から消す

というステップを牛の数だけ繰り返し, 全頭まで配置できれば成功として判定できる.

問題	Water Tank	(模擬地区予選 2009)
タンクの容量と時間毎の水の使用量が与えられるので, 最低限必要な給水速度を求める. (何周してもギリ貧にならない量が求められている) (同名の別問題もあるので注意) http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2180&lang=jp		

ある給水速度を仮定した時に, それで生活できるかどうかを判定する関数を `ok` とする. 求める速度を含む区間を $[lo, hi]$ で表現し, 生活できるか出来ないかで範囲を半分に狭めてゆく. 区間が十分狭くなったら終了して出力する. 考え方と効果は二分探索 (binary search) とほぼ同様だが, 連続区間を扱う場合は二分法 (bisection) と呼ばれることが多い.

```
C++
1  double lo = sum/86400, hi = 1e6; // 1e6 = 10^6
2  while (hi-lo>1e-6) {
3      double m = (hi+lo)/2.0;
4      if (ok(m)) hi = m;
5      else lo = m;
```

```

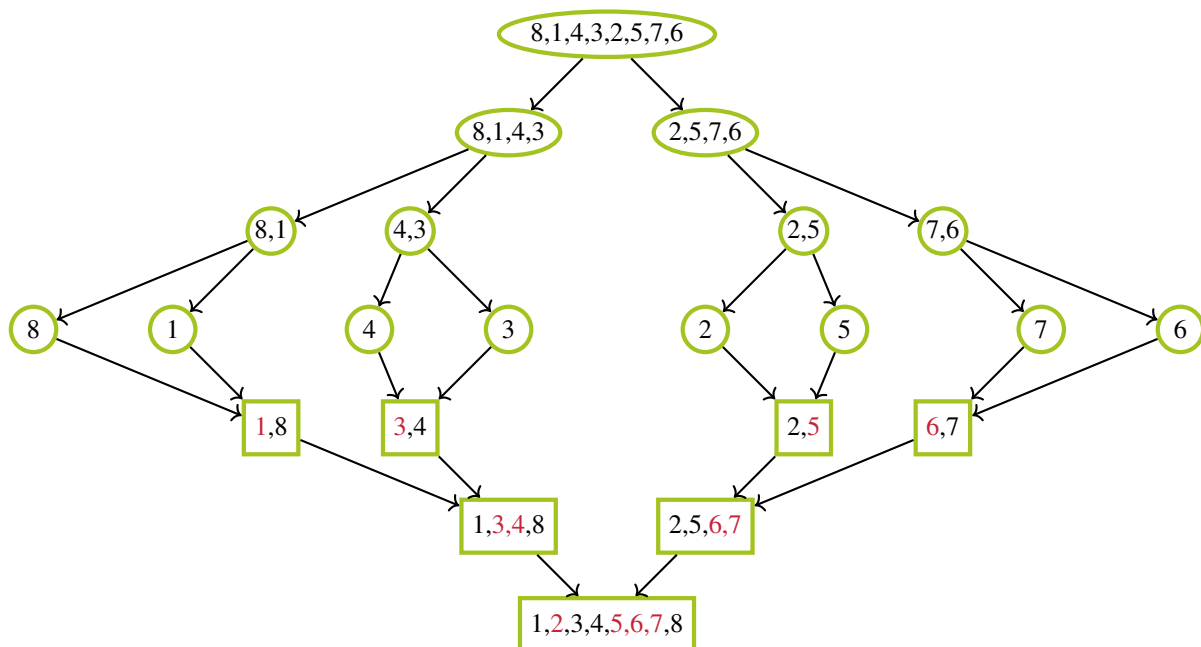
6     }
7     printf("

```

原理は単純なのだが、関数 `ok` の実装においてタンクの最大容量以上は水を貯められないなどの処理で慎重さが必要な点があるので、次の節の `inversion count` の問題のほうが易しいかもしれない。

5.2 Merge Sort

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



5.2.1 Inversion Count

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

```

C++ 1  int N, A[128];
    2  int solve() {
    3      int sum = 0;
    4      for (int i=0; i<N; i++)
    5          for (int j=i+1; j<N; j++)

```

```

6         if (A[i] > A[j]) ++sum;
7     return sum;
8 }

```

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

```

C++ 1 int N, A[たくさん]; // A は元の配列
    2 int W[たくさん]; // W は作業用配列
    3 int merge_and_count(int l, int r) { // range [l, r)
    4     if (l+1 >= r) return 0; // empty
    5     if (l+2 == r) { // [l, r) == [l, l+1] 要素 2 つだけ
    6         if (A[l] <= A[l+1]) return 0; // 逆転はなし
    7         swap(A[l], A[l+1]);
    8         return 1; // 逆転一つ
    9     }
   10     int m = (l+r)/2; // [l, r) == [l, m) + [m, r)
   11     int cl = merge_and_count(l, m); // 左半分を再帰的に数える
   12     int cr = merge_and_count(m, r); // 右半分を再帰的に数える
   13     int c = 0; // 左と右を混ぜるときの逆点数
   14     int i=l, j=m; // i は [l, m) を動き, j は [m, r) を動いて,
   15     int k=l; // 小さいものから W[k] に書き込む
   16     while (i<m && j<r) { // A[i] と A[j] を比べながら進む
   17         if (A[i] <= A[j]) W[k++] = A[i++]; // 左半分の方が小さく逆転なし
   18         else {
   19             W[k++] = A[j++];
   20             c += XXX; // 左半分の方が大きい, 左半分で未処理の要素だけ飛び越える
   21         }
   22     }
   23     while (i<m) W[k++] = A[i++]; // 左半分が余った場合
   24     while (j<r) W[k++] = A[j++]; // 右半分が余った場合
   25     assert(k == r);
   26     copy(W+l, W+r, A+l);
   27     return cl + cr + c;
   28 }

```

5.2.2 練習問題

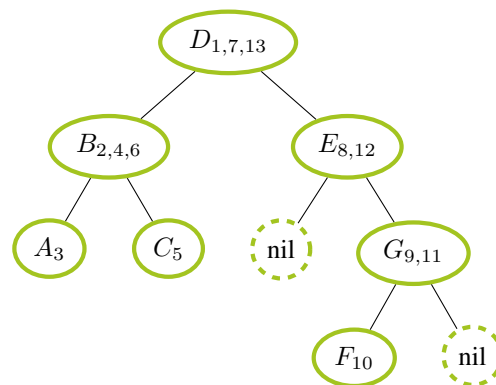
問題	Recursion / Divide and Conquer - The Number of Inversions (AOJ)
<p>与えられた数列内の、大小関係が逆転しているペアの個数を求める</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D&lang=jp</p>	

問題	Ultra-QuickSort	(Waterloo local 2005.02.05)
類題: 数が大きいので long long を使うこと. http://poj.org/problem?id=2299		

問題	Japan★	(Southeastern Europe 2006)
長方形の島の東西に道路が走っている. 交わる箇所数を求めよ http://poj.org/problem?id=3067		

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

5.3 木のたどり方



例 1: 数字は訪問順序を表す

根から始めて, 下のような再帰的な手続き (深さ優先探索, 8 章) で, 辺をたどりながら各頂点を一巡することを考える:

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

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

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

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

問題

Tree - Reconstruction of a Tree

(AOJ)

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

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_7_D&lang=jp

問題

Tree Recovery*

(Ulm Local 1997)

上に同じ (複数ケースが与えられる点と, 書式が少し異なる)

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

ヒント:

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

回答例:

C++

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  string preorder, inorder;
5  // preorder の [fp,lp) の範囲と, inorder の [fi,li) の範囲について
6  // 木を postorder で表示
7  void recover(int fp, int lp, int fi, int li) {
8      int root;
9      // preorder[fp] == inorder[root] となるような root を求める
10     if // (左側が存在すれば)
11         recover(fp+1, fp+(root-fi)+1, fi, root); // 左側を表示
12     if // (右側が存在すれば)
```

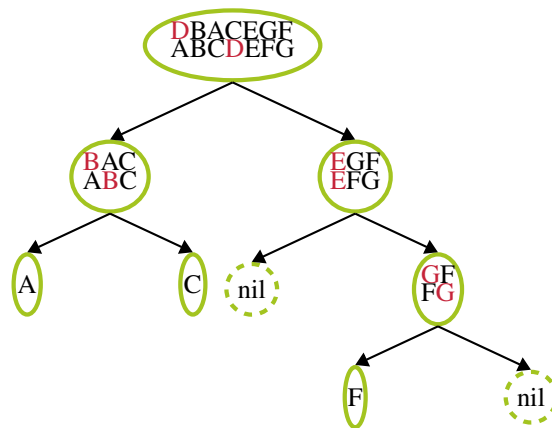


```

13         recover(fp+(root-fi)+1, lp, root+1, li); // 右側を表示
14     cout << inorder[root]; // root を表示
15 }
16 int main() {
17     while (cin >> preorder >> inorder) {
18         recover(0, preorder.size(), 0, inorder.size());
19         cout << endl;
20     }
21 }

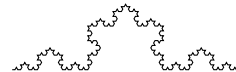
```

呼び出し関係を図にすると、以下のようになる。図中、赤字が(その時点での部分木における)根を表す。頂点内の文字列は、その頂点以下の部分木に対する、preorder 表記 (上段, fp と lp の範囲) と inorder 表記 (下段, fi と li の指す範囲)。



5.4 空間充填曲線

問題	Recursion / Divide and Conquer - Koch Curve	(AOJ)
<p>コッホ曲線の頂点を計算せよ。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_C&lang=jp</p>		



ヒント: 基準線が (x_0, y_0) から $(x_0 + dx, y_0 + dy)$ に引かれるとき, 左手側にこぶを作るには, 中点 $(x_0 + dx/2, y_0 + dy/2)$ から, $(-dy, dx)$ 方向に適当な距離を伸ばせば良い。

コッホ曲線を描く再帰関数の設計には 2 通りの方針がある: 一つは表示させる設計, もう一つは点列を返す設計である。

関数 `koch(x0, y0, x1, y1, n)` を, 「座標 (x_0, y_0) から (x_1, y_1) までの点を表示する (端点は (x_0, y_0) を含み (x_1, y_1) を含まない)」と設計すると, 以下のような構造になる:

```

Python3 1 import math

```

```

2 def show_vertex(x,y):
3     print("
4 def_koch(x0,y0,_x1,y1,_n):
5     if _n==0:
6         show_vertex(x0,y0)
7     else:
8         ... # 中間点 c1,c2,c3 を計算
9         koch(x0,y0,_c1[0],c1[1],_n-1)
10        koch(c1[0],c1[1],_c2[0],c2[1],_n-1)
11        koch(c2[0],c2[1],_c3[0],c3[1],_n-1)
12        koch(c3[0],c3[1],_x1,y1,_n-1)
13
14 N=_int(input())
15 koch(0,0,100,0,N)
16 show_vertex(100,0)

```

点列を返す場合は、以下のように書ける (全体の計算終了後に最後にまとめて表示する):

Python3

```

1 def koch(x0,y0, x1,y1, n):
2     if n == 0:
3         return [(x0,y0)]
4     else:
5         ... # 中間点 c1,c2,c3 を計算
6         return koch(x0,y0, c1[0],c1[1], n-1) \
7             + koch(c1[0],c1[1], c2[0],c2[1], n-1) \
8             + koch(c2[0],c2[1], c3[0],c3[1], n-1) \
9             + koch(c3[0],c3[1], x1,y1, n-1)
10
11 def show_vertex(x,y):
12     print("
13
14 N=_int(input())
15 V=_koch(0,0,100,0,N)+[[100,0]]
16 for _x,_y in V:
17     show_vertex(_x,_y)

```

C++ では、後者の設計の実装は面倒があるので、前者の設計 (再帰関数の中で表示) が簡便である。



点列の図示

答えが合わないときや、方針に悩んだ時は点列を図示すると良い。gnuplot が利用可能な環境では、次の手順で単に図示できる。(1) ターミナルで gnuplot を起動する (2) gnuplot のプロンプトで plot 'xy.txt' with lines とタイプする。ただし、プログラムが出力した座標を "xy.txt" に保存したとする。

ま

```

$ gnuplot 1
gnuplot> plot 'xy.txt' with lines 2

```

た Python の場合は, matplotlib を使うと便利である.

```
Python3 1 import matplotlib.pyplot as plt
2 import numpy as np
3 def plot_koch(a):
4     fig = plt.figure()
5     ax = fig.add_subplot(1,1,1)
6     a_ = np.array(a)
7     xlist = a_[:, 0]
8     ylist = a_[:, 1]
9     ax.plot(xlist, ylist)
10    ax.set_title('koch_curve')
```

(jupyter 内で表示するには, 先頭で `%matplotlib inline` を実行しておく)

問題	Riding the Bus★★	(世界大会 2003)
<p>正方形内に描かれた Peano curve 上に格子点がある. 与えられた二点間の距離を求めよ. 距離とは, 与えられた点から最短の格子点への距離 (複数ある場合は x,y 座標が小さいものまで) と, 格子点間の Peano curve 上の道のり. (誤差の記述がちょっと心配)</p> <p>https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=37&page=show_problem&problem=724</p>		

解き方の例: 全体を 9 分割して, 関係ある場所だけを細かく探す.

問題	Plotter★★	(Algorithmic Engagements 2011)
<p>与えられた点をいつペンが通るか.</p> <p>http://main.edu.pl/en/archive/pa/2011/plo</p>		

第 6 章

基本データ構造 (string, stack, queue, string, set, map)

概要

この章では、C++ や Java など多くの言語で標準ライブラリとして提供されている「道具」として、文字列、スタック、(優先度付き) キュー、集合、連想配列を紹介する。道具の紹介が退屈な読者は、一旦飛ばして先に進んでおいて、あとで必要になってから戻ってくる読み方も可能である。なお、この資料では、まず標準ライブラリを使いこなすことを勧める立場を取る。これらの道具を自作することも筋力トレーニングとしては有用であるが、初学者には向かない。また実用的な状況では、標準ライブラリで用が足りるならその方が望ましい。たとえば、多くの人に使われているのでバグがないなどのメリットが期待される、他の人も性質を良く知っているため共同作業に向いている、などの利点がある。

なお、この章では、C++ と Python3 についてサンプルコードを取り上げる。Ruby については、付録 C を参照。

6.1 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。

6.2 スタック (stack) とキュー (queue)

スタックとキュー (参考書 ^{攻略}[2, pp. 80–], 参考書 [3, pp. 31–]) は、データを一時的に保存したり取り出した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。

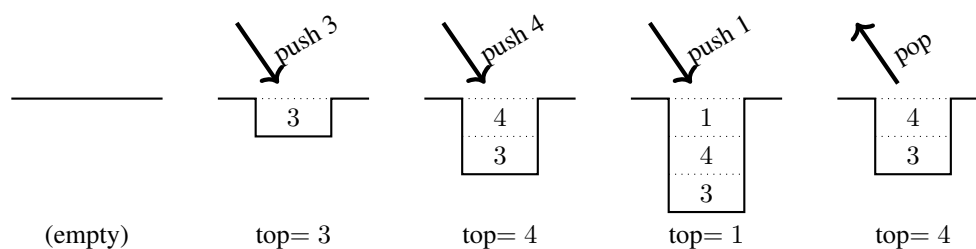
優先度付きキュー では、早く並んだ順ではなく各データの優先度の高い順に取り出される。現実にはぴったりした例がないが、待っている人の中から優先度の高い順に診療が行われるような状況や、高い買値を提示した順に品物を入手できる市場などがもしあれば相当するだろう。

なお、C++ の標準ライブラリで提供されているでは取り出し操作が、先頭要素の参照 `top()` あるいは `front()` と先頭要素の削除 `pop()` という二つの操作に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。

6.2.1 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。

^{*1} これは exception safety のためである。



■使用上の注意 通常は、配列や `vector` などが用いられる。典型的な実装での計算コストは `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、各操作の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ではなく、両者が分離されていることに注意^{*2}。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。

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^{*3}。

C++

```
1 #include <stack>
2 #include <iostream>
3 using namespace std;
4 int main() {
5     stack<int> S; // int を格納するスタック
6     S.push(3); // 先頭に追加
7     S.push(4);
8     S.push(1);
9     while (!S.empty()) { // 要素がある間
10         int n = S.top(); // 先頭をコピーして
11         S.pop(); // 先頭要素を廃棄
12         cout << n << endl; // 1, 4, 3 の順に表示される
13     }
14 }
```

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

Python3

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

^{*2} これは exception safety のためである。

^{*3} ウェブにも資料がある <http://en.cppreference.com/w/cpp/container/stack>

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。参考書 ^{攻略}[2, p.82] に詳しい解説が掲載されている。

C++

```

1  #include <string>
2  #include <stack>
3  #include <iostream>
4  using namespace std;
5  int main() {
6      string word;
7      stack<int> S;
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 n に変換するには、C++11 の場合は `int n=stoi(word)` を、そうでない場合は `<cstdlib>` を include して、`int n=atoi(word.c_str())` などとする。(AOJ の C++11 はまだ `stoi` をサポートしていないので `atoi` を用いる)

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)

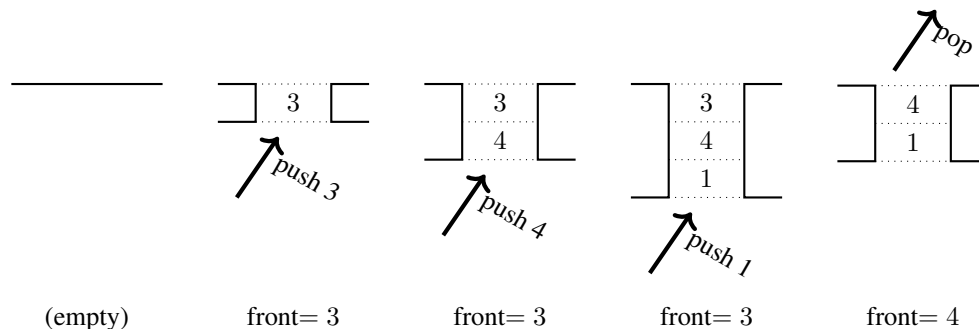
<http://poj.org/problem?id=2559>

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>

6.2.2 キュー

■概要 キュー (queue, 参考書 [3, p. 32]) 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。



■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、各操作の計算時間は増えないと期待される。(データ全体を移動させると $O(N)$ になってしまうので、実際には head と tail のような現在使用中の区間をあらわす添字またはポインタのみを操作する)。

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。^{*4}

```
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 }
```

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。^{*5}

Python3

^{*4} <http://en.cppreference.com/w/cpp/container/queue>

^{*5} <https://docs.python.jp/3/library/collections.html#deque-objects>


```

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)

複数の計算を少しず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。参考書^{攻略}[2, p.82] に詳しい解説が掲載されている。

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&lang=jp

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、

-
-

とすることで、同じ高さの下りと上りのペアをつくる事が出来る。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

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。

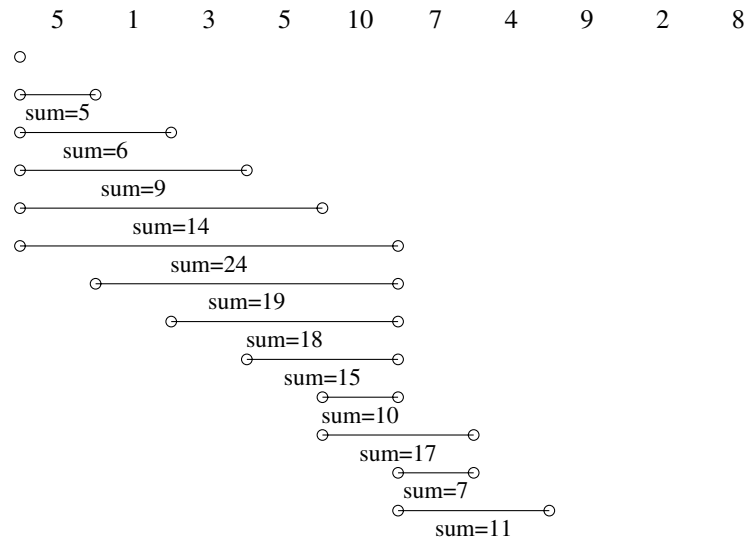
<http://poj.org/problem?id=3061>

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

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&lang=jp>

初めにエラトステネスの篩 (参考書 [3] 112 ページ) などの手法で、10 000 までの素数を求めておく。あとは “Subsequence” と同じ。

6.2.3 優先度付きキュー

優先度付きキュー (priority queue, 参考書 [3, p. 32]) 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。

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&lang=jp

まずは、標準ライブラリ `priority_queue`^{*6}を用いて、`accepted` を得られることを確認すると良い。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。詳しくは参考書^{攻略}[2, 10 章, pp. 232–] を参照。

```
C++
1  #include <queue>
2  #include <iostream>
3  using namespace std;
4  int main() {
5      priority_queue<int> Q; // int を格納する優先度つきキュー
6      Q.push(3); // 追加
7      Q.push(4);
8      Q.push(1);
9      while (!Q.empty()) { // 要素がある間
10         int n = Q.top(); // 先頭をコピーして
11         Q.pop(); // 先頭要素を廃棄
12         cout << n << endl; // 4, 3, 1 の順に表示される
13     }
14 }
```

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
1  import heapq
2
3  Q = []
4  heapq.heappush(Q, 3)
5  heapq.heappush(Q, 4)
6  heapq.heappush(Q, 1)
7  while len(Q) > 0:
8      n = heapq.heappop(Q)
9      print(n) # 1, 3, 4 の順に表示される
```

6.3 文字列と分割・連結・反転

C++ では `string`^{*7}データの部分文字列を `substr` メソッド^{*8}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  }
```

^{*6} http://en.cppreference.com/w/cpp/container/priority_queue

^{*7} http://en.cppreference.com/w/cpp/string/basic_string

^{*8} http://en.cppreference.com/w/cpp/string/basic_string/substr

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
```

反転には, `reverse` という関数を用いる. 範囲は `word.begin()` と `word.end()` で指定する. 配列の場合と近い記法で, `reverse(&word[0], &word[0]+word.size());` と書くこともできる.

```
C++ 1 #include <algorithm>
2 string word = "hello";
3 reverse(word.begin(), word.end());
4 cout << word << endl;
```

6.4 集合 (set)

C++ で集合を表現するために標準ライブラリに `set`^{*9} 及び `unordered_set`^{*10} (C++11 以降) というデータ構造が標準が用意されている. 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.

^{*9} <http://en.cppreference.com/w/cpp/container/set>

^{*10} http://en.cppreference.com/w/cpp/container/unordered_set

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      typedef set<string> set_t;
4      set_t A;
5      cout << A.size() << endl; // 初めは空なので 0
6      cout << A.count("hello") << endl; // "hello"は含まれていないので 0
7      A.insert("hello");
8      A.insert("world");
9      A.insert("good_morning");
10     A.insert("world");
11     cout << A.size() << endl; // "hello", "good morning", "world"
12     cout << A.count("world") << endl; // 1 個 set の場合は最大 1
13     cout << A.count("hello!!!") << endl; // 0 個

```

python の場合は set が用意されている。^{*11}

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

^{*11} <https://docs.python.jp/3/library/stdtypes.html#set>



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。

6.4.1 bitset

集合の特殊な場合として、0 から N までの整数の集合を扱う場合は、ビット表現を用いると効率が良い。C++ では標準で bitset 型が用意されている。

```
C++
1 #include <bitset>
2 #include <assert>
3 bitset<128> a, b, c;
4 a[3] = 1; // 指定した bit を on に
5 b[5] = 1;
6 c = a | b; // 集合の和
```

6.5 連想配列 (map)

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。

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

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`^{*12} と `unordered_map`^{*13}, Python では `dict`^{*14} が標準で用意されている。

6.5.1 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
```

*12 <http://en.cppreference.com/w/cpp/container/map>

*13 http://en.cppreference.com/w/cpp/container/unordered_map

*14 <https://docs.python.jp/3/library/stdtypes.html#dict>

```

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("
9     #_good_morning_20
10    #_hello_1
11    #_world_100

```

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>

string 型の文字列 s の長さは $s.size()$ で入手することができる。

6.5.2 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

```

```

Python3 1 # 普通の hash の利用
2 table = {}
3 # table["ichirou"] += 1 ... ng!
4 table["ichirou"] = table.get("ichirou", 0) + 1
5 if not "world" in table:
6     table["world"] = 1

```

```

7  # Counter の利用
8  import collections
9  c = collections.Counter()
10 c["ichirou"] += 1 # ok

```

6.5.3 連想配列の要素の一覧

C++11

```

1  map<string,int> phone;
2  phone["taro"] = 123;
3  phone["jiro"] = 456;
4  phone["saburo"] = 789;
5
6  for (auto p=phone.begin(); // i=0 に相当
7       p!=phone.end(); // i<N に相当
8       ++p) {
9      cout << p->first << " " << p->second << endl;
10 }
11 // 出力順は
12 // jiro 456
13 // saburo 789
14 // taro 123

```

赤字で書いた部分が定型句であるが、初見では把握が難しいため、そのまま用いてほしい。配列の場合は添字として整数 i を用いるが、連想配列の場合は複雑な処理があるため、整数の代わりにイテレータと呼ばれる抽象インターフェースを用いる。^{*15} また初期値と終了値も、0 や N の代わりに `begin()` と `end()` を用いる。ループの中で、各要素の `key` は `p->first`, `value` は `p->second` で参照する。

6.6 練習問題

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>

C++

```

1  int M;
2  string S;
3  int main() {
4      cin >> M;
5      for (int i=0; i<M; ++i) {
6          cin >> S;

```

^{*15} ここでは C++11 の機能を利用して、`auto` と書いた。実際の方は `map<string,int>::iterator` である。


```

7      ... all; // 文字列の集合
8      ... // all に S を挿入
9      for (size_t i=1; i<S.size(); ++i) {
10         string L;
11         ... // L を S の [0, i) 文字に設定
12         string R;
13         ... // R を S の [i, S.length()) 文字に設定
14         string L2, R2;
15         ... // L2 を L の逆順に設定
16         ... // R2 を R の逆順に設定
17         ... // all に R+L, R2+L, L+R2.. 色々挿入
18     }
19     ... // all の要素数 (all.size()) を出力
20 }
21 }

```

問題	Era Name	(模擬地区予選 2010)
<p>(架空の) 西暦と (架空の) 元号の対応が与えられるので、整理して記憶する。続いて、西暦が質問として与えられるので、元号がわかっている元号を、そうでなければ “Unknown” と答える。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2242&lang=jp</p>		

問題補足: 入力例の “showa 62 1987” は、

- 昭和元年 (1 年) の西暦は、1926 年である
- 昭和は、(少なくとも) 1987 年まで続いた

という二つの情報をあらかず。他に情報がなければ、1988 年が昭和であるかは分からない (昭和と平成の間に別の元号が使われているかもしれない) ので、 “Unknown” と答える。

回答例: 各情報を、西暦とその年が 1 年である元号の対応表と、元号が最大何年まで続いたかの二つに分けて記録する。前者には `map<int, string>` を、後者には `map<string, int>` を用いる。

6.6.1 区間の管理

問題	Restrictive Filesystem*	(模擬国内予選 2009)
<p>単純な管理を行うファイルシステムでの、読み書きと消去を実装し、各時点で読まれるデータを答えよ。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2152&lang=jp</p>		

回答例:

6.6.2 集合の応用

問題	Twenty Questions★	(アジア地区予選 2009)
<p>それぞれの object を区別するのに、最善の質問の仕方を考える。答え次第で次の質問を変えて良い。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1302&lang=jp</p>		

問題	Sweets★	(Algorithmic Engagements 2010)
<p>正の数が $n \leq 24$ 個ある。兄弟 A, D, B でそれらを分ける。$A \geq D \geq B$ を満たす分け方の中で、$A - B$ が最小の分け方を探す。</p> <p>https://szkopul.edu.pl/problemset/problem/cEMM4bdcKN06uikzs9BRSqkz/site/</p>		

解説されている解法を担当者が実装したつもののもので tle なので (最大ケース 5 秒くらい)，“Useful Resources” からデータをダウンロードして正しければ時間は気にしないことが良さそう。

問題	Building Blocks★	(15th Polish Olympiad in Informatics)
<p>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。</p> <p>https://szkopul.edu.pl/problemset/problem/KC7c6nYfAXCbCGszqhIeOGxP/site/</p>		

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。

第 7 章

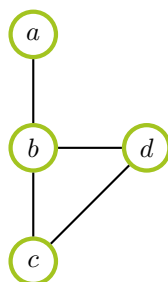
グラフ (1) 全域木

概要

本日から複数回、グラフの話題を扱う。グラフは、様々な状況や問題をモデル化するための、基礎となる概念である。今週は、最小重み全域木を求めてみよう。またそのための道具として、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。

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。(参考書 [3, 2-5 節, p. 87] あれもこれも実は“グラフ”)



7.1.1 グラフに関する用語

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

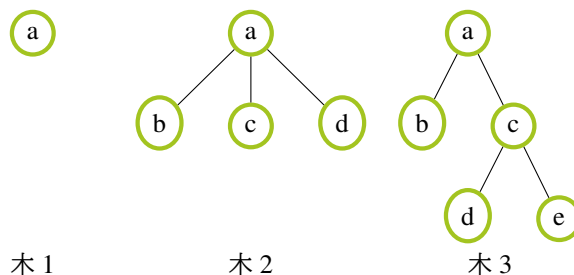
例: 3 つの頂点 $V = \{1, 2, 3\}$ の辺を全て結んだグラフ (=三角形) は, $E = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$

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

例: 三角形の各頂点の次数は 2

7.1.2 木

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



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

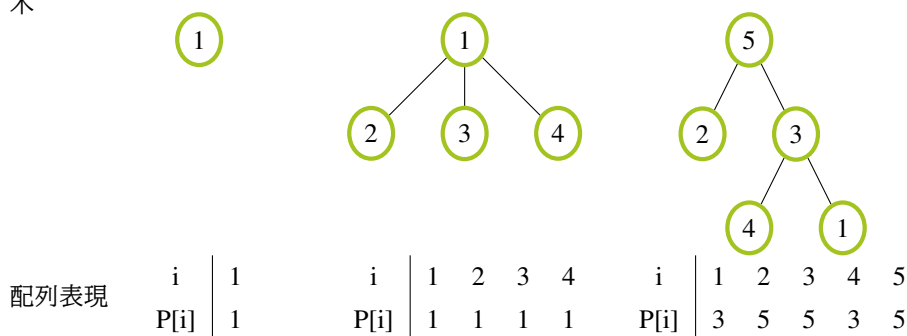
木の頂点の特別な一つを根 (root) と呼ぶ。グラフを図示する際には、根を一番上または下に配置することが多い。次数 1 の点を葉 (leaf) と呼ぶ。木の辺で結ばれた 2 頂点について、根に近い方の頂点 (あるいは根) を親 (parent) と呼び、遠い方を子 (child) と呼ぶ。

7.1.3 「親」に注目した、木の表現

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

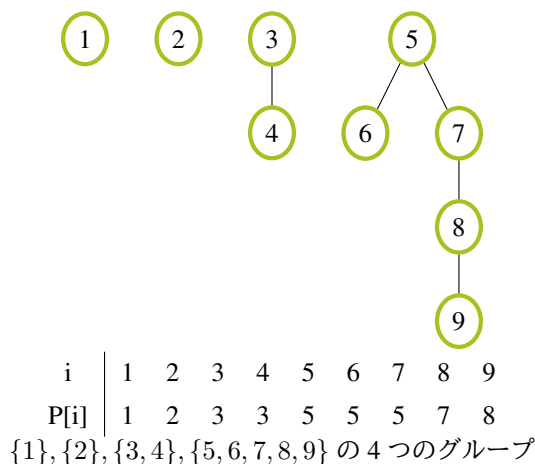
なお、木ではない一般のグラフでは、各節点で子を複数持ちうるので、子を管理する場合は一次元配列より複雑な表現 (隣接リスト、隣接行列など 8.1) が必要となる。

木



7.2 Disjoint Set (Union-Find Tree)

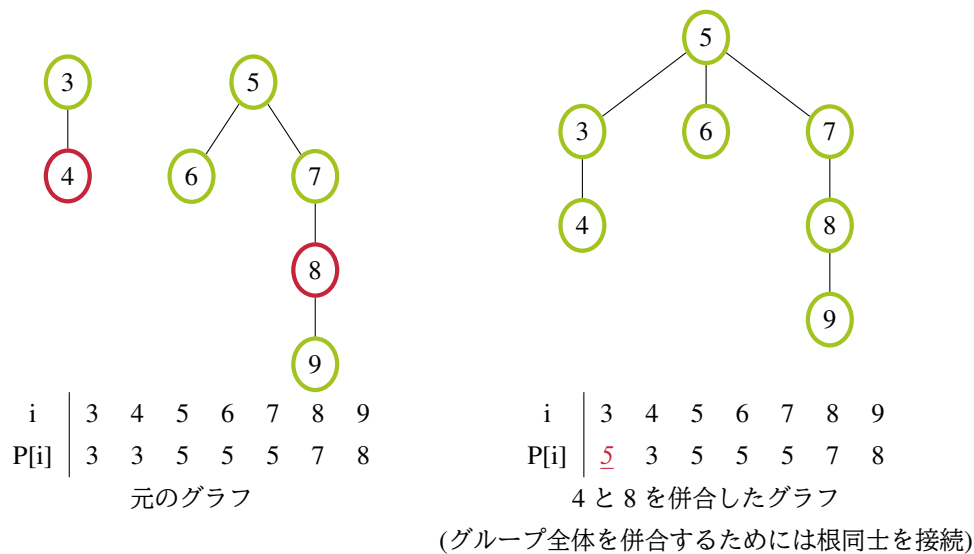
二つの要素が同一グループに属するかどうかを効率的に判定する手法の一つが、Union Find 木 (参考書 攻略[2, pp. 318–], 参考書 [3, pp. 81–]) である。Disjoint-set とも呼ばれる。グループ同士を併合する操作ができるが、分離はできない (c.f. link-cut tree).



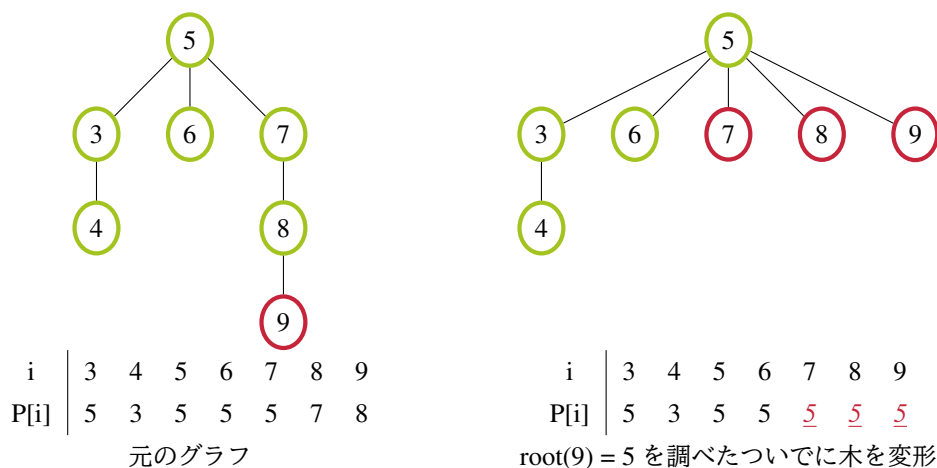
■判定 同じ木に属するなら同じグループで、そうでなければ異なるグループである。それを、根 (root) を調べることで行う。例:

- Q1. 6 と 8 は同じグループに属するか?
6 の属する木の根 (root) は 5 で、同様に 8 の根は 5 → 同じグループ.
- Q2. 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、意味は変わらない。(小さな (低い) 木を大きな (高い) 木の下につける方が効率が良い、が、この資料では割愛する)



■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、一度根を調べたら、関係する接点になるべく根に直接つなげるようにつなぎ替えると効率の改善に有効である。



■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、一度書き写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++
1 int P[10010]; // 0 から 10000 までの頂点を取り扱い可能
2 void init(int N) { // 初期化 はじめは全ての頂点はバラバラ
```

```

3     for (int i=0; i<N; ++i) P[i] = i;
4 }
5 int root(int a) { // aのroot(代表元)を求める
6     if (P[a] == a) return a; // aはroot
7     return (P[a] = root(P[a])); // aの親のrootを求め, aの親とする
8 }
9 bool is_same_set(int a, int b) { // aとbが同じグループに属するか?
10    return root(a) == root(b);
11 }
12 void unite(int a, int b) { // aとbを同一グループにまとめる
13     P[root(a)] = root(b);
14 }

```

関数 root の最後で $P[a]$ に代入している部分が、パス圧縮である。

```

Python3 1 P = [i for i in range(N)]
2 def root(x):
3     path_to_root = []
4     while P[x] != x:
5         path_to_root.append(x)
6         x = P[x]
7     for node in path_to_root:
8         P[node] = x # パス圧縮
9     return x
10 def is_same_set(x, y):
11     return root(x) == root(y)
12 def unite(x, y):
13     P[root(x)] = root(y)

```

Python では、再帰の深さの制限が C++ よりきついのので、関数 root を再帰ではなく while で実装しておく。
実行例:

```

C++ 1 int main() {
2     init(100);
3     cout << is_same_set(1, 3) << endl;
4     unite(1, 2);
5     cout << is_same_set(1, 3) << endl;
6     unite(2, 3);
7     cout << is_same_set(1, 3) << endl;
8 }

```

例題

Disjoint Set: Union Find Tree

(AOJ)

Disjoint Set でグループを管理せよ

注意: S_1, \dots, S_k の集合の添字は無視すること. とくに「 x を含む集合 S_x 」の S_x と対応するわけではない.

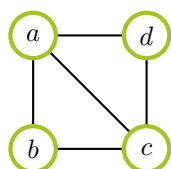
(例題 続き)

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

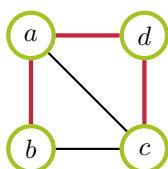
7.3 全域木

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。

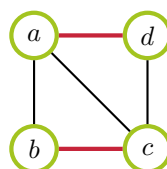
メモ: グラフが連結なら全域木が存在する。全域木は一つとは限らない (完全グラフの場合は n^{n-2} 個もある)。最小全域木も一つとは限らない。最小全域木を求める問題は、最小全域木のうちの一つを求める問題を指すことが多い。



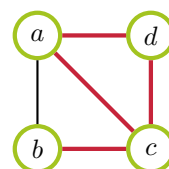
元のグラフ



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)

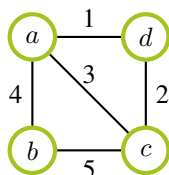
7.3.1 クラスカル法

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

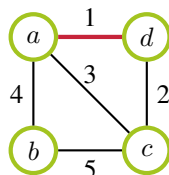
クラスカル法は以下のように動作する (参考書 [3, pp. 101–]):

- 辺を重みの小さい順にソートする (数字の組をソートする方法は 3.1.3 章を参照)
 - 方法 1: $\langle \text{重み}, \text{始点}, \text{終点} \rangle$ をソート
 - 方法 2: $\langle \text{重み}, \text{辺 ID} \rangle$ をソートし、ID と始点や終点を対応させる配列を別に管理
- T を作りかけの森 (最初は空、閉路を含まないグラフ、連結とは限らない) とする
- 重みの小さい順に、各辺 e に対して以下の操作を行う

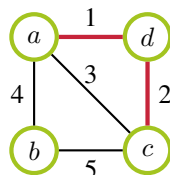
$T + e$ (グラフ T に辺 e を加えたグラフ) が閉路を含まなければ T に e を加える



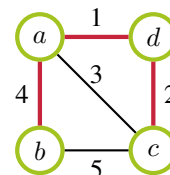
元のグラフ



辺 ad (重み 1) を採用



辺 cd (重み 2) を採用



辺 ab (重み 4) を採用

$T + e$ が閉路を含むかどうかを効率的に判定する手法の一つが、union-find tree である。上記のクラスカル

法内で、辺 e を加える度に `unite` で辺の両端の頂点を同一グループに組み込む。辺 e の両端を a, b として、`is_same_set(a, b)` が真であれば辺 e を加えると閉路ができる (e と別に a, b パスがあるので)。

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&lang=jp

問題

Stellar Performance of the Debunkey Family

(PC 甲子園 2008)

問題: 街を連結に保ったまま、橋の維持費用を最小化したい。

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

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

C++

```
1 #include <algorithm>
2 int N, M, A[10010], B[10010], COST[10010];
3 pair<int,int> bridge[10010]; // コストと橋番号のペア
4 int main() {
5     while // (N と M を読み込み, N が 0 より大きければ処理する)
6         for (int i=0; i<M; ++i) {
7             // A[i] と B[i] と COST[i] を読み込む;
8             bridge[i].first = COST[i];
9             bridge[i].second = i;
10        }
11        sort(bridge, bridge+M); // コストの小さい順に整列
12        for (int i=0; i<M; ++i) {
13            int cost = bridge[i].first;
14            int a = A[bridge[i].second];
15            int b = B[bridge[i].second];
16            // 「a から b に cost の橋がかかっている」と表示
17        }
18    }
19 }
```

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

■回答例: 上記の準備ができているとして、回答の骨子は以下ようになる。

C++

```
1 int 合計 = 0;
```

```

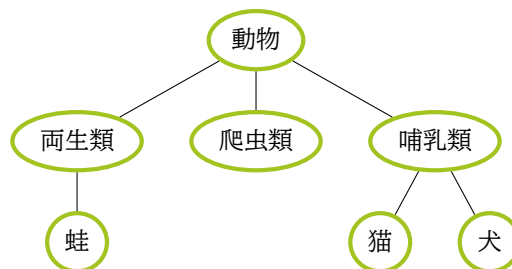
2      for (int i=0; i<M; ++i) { //安い橋から順に
3          if // (端の両端の節点が既に連結だったら)
4              continue;
5          // 橋の両端の節点を同一グループに併合する;
6          // 合計に、橋のコストを加える;
7      }
8      // 合計を出力;

```

7.4 練習問題

問題	Marked Ancestor	(夏合宿 2009)
<p>マークされている最も近い祖先を求める。初めは根だけがマークされている。</p> <p>注意: 合計は int を越えるので, long long を用いる。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2170&lang=jp</p>		

補足: 下記の図で、初めに「猫」の“nearest marked ancestor”は「動物」、もし「哺乳類」がマークされた後なら、「猫」の“nearest marked ancestor”は「哺乳類」。



ヒント

まず、Mark や Query などの命令列を全て読み込み、

7.4.1 Disjoint Sects

問題	Fibonacci Sets	(会津大学プログラミングコンテスト 2003)
<p>i 番目の Fibonacci 数を $f[i]$ と表記する。ある i, j ($1 \leq i, j \leq V$) について、$f[i]\%1001$ と $f[j]\%1001$ の差の絶対値が d 未満だったら、ノード i と j は同じグループに属するとする。 V と d が与えられた時に、グループの数を数えよ。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1016&</p>		

(問題 続き)

lang=jp

回答例:

```

C++
1  int F[1001];
2  int main() {
3      F[0] = 1, F[1] = 2;
4      // ... F[i] に i 番目の Fibonacci 数 %1001 をあらかじめ計算, 代入しておく
5      while (cin >> V >> D) {
6          // 木を初期化
7          for (int i=1; i<=V; ++i)
8              for (int j=i+1; j<=V; ++j)
9                  if // (F[j] と F[i] の差の絶対値が D 未満だったら)
10                     // グループ i とグループ j を同一化;
11                 // i ∈ [1, V] に関して root(i) == i であるような数を数えて出力
12            }
13 }

```

細かく作ってテストする:

- Fibonacci 数の計算: $F[2]..F[1000]$ までを for 文で代入する (12.4.1 の方針 3 で $a[2]$ の代わりに F を用いることと, 1001 の剰余を取りながら行う点が差). つまり, i を小さい方から大きくしてゆけば, 定義通りに $F[i] = F[i-2] + F[i-1]$; と計算できる. オーバーフローしないように計算途中でも 1001 の剰余を取る. (表示して確認する)
- Union-find 木を作る: 基本的には資料通り
(初期化して, 木を表示して確認する. いくつか併合しては木を表示して確認する.)
- Fibonacci 数での動作作成: 小さな V (たとえば 10) と適当な D を与えて, 木を表示し, 手計算と一致するかどうかをテストする.
- グループの数を数える: $root(i) == i$ である個数ををテストする.

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&lang=jp>



ヒント

正直者かどうかは直接的には分からなくても、

問題

Never Wait for Weights*

(アジア地区予選 2012)

(意識) 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&lang=jp>

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

問題

Everlasting -One-*

(アジア地区予選 2013)

転職できる仕事をグループにして、行き来できないグループがどれだけあるかを知りたい。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2568&lang=jp>

7.4.2 色々な全域木

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&lang=jp>



ヒント

全域木の最大の辺の重みについて考える。クラスカル法は、

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&lang=jp		

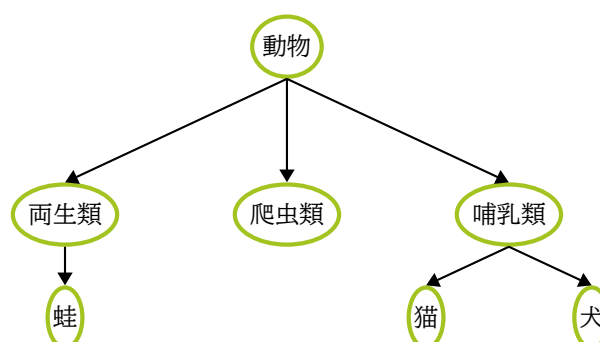
問題	Byteland★	(1st Junior Polish Olympiad in Informatics)
同じ重みの辺が存在する場合は, 最小重み全域木は複数通り存在しうる. 各辺毎に minimum spanning tree に含まれるかを調べる https://szkopul.edu.pl/problemset/problem/JXaPu39wQfiZaqlM51zlchez/site/		

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

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		

7.5 補足: 木に関連する他の話題

7.5.1 最小共通祖先



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

例題

Nearest Common Ancestors

(Taejon 2002)

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

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

いくつかのステップに分割して解いてみよう:

1. 入力を目で理解する。一行目に数字 T があり、「木の情報と LCA を求める節点二つ」が T セット続いている。Sample Input の木を紙に書いてみる。
2. 木の入力を読み込む各 $P[]$ を表示して確認せよ。各節点の親は、紙に描いた木と一致するか?
3. LCA を求める準備として、ある節点 x から根までの節点を全て求めることを考える。考え方としては、 x の親は $P[x]$ で求められるから、親の親、親の親の親などとたどっていけばいずれは根に到達する。親をたどる際に節点を表示して、確認してみよう。
4. 節点 A と B のそれぞれについて、根までのパス (頂点番号列) の共通要素を求める。一番根から遠いものが求める答え。

なお、発展的な話題になるが、一つの木について何度も LCA を求める場合には、事前に準備をしておく一回あたり $O(\log N)$ と効率的に求められる (16.3 章や参考書 [3, p. 274] を参照)。

問題

Apple Tree*

(POJ Monthly-2007.08.05)

りんごの数を数える

<http://poj.org/problem?id=3321>

7.5.2 親への移動

問題

Marbles on a tree

(Waterloo local 2004.06.12)

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

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

入力のサンプルコードを示す:

C++

```
1 #include <cstdio>
2 using namespace std;
3 int N, P[10010], M[10010], V[10010];
```

```

4  int main() {
5      while (~scanf("
6  %d",&P,&P+N,&L-1);
7  %d",&id,&c;
8  %d",&i,&i<N; ++i) {
9      scanf("
10     %d",&id);
11     for (int j=0; j<V[id]; ++j) {
12         scanf("
13         %d",&c);
14         P[c] = id;
15     }
16     if (V[id] == 0) ++L; // Vが0ならidは葉
17     for (int i=0; i<N; ++i) {
18         // 親を出力してみる
19         printf("
20 %d\n", P[i]);
21     }
22 }

```

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

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

C++

```

1  // 節点が調整済みかを表す配列を用意する。初めに葉を調整済みとしておく。
2  // 各節点での過不足を表す配列を用意する。初期値は、果物の配置数-1
3  while // 根が調整済みでない
4      for // 全ての節点 i について
5          if // i が調整済みでない かつ i の子供が全て調整済みなら
6              // i での過不足を親に押し付ける
7              // i を調整済みと記録
8          }
9      }
10 }
11 // 各節点での過不足の合計が答え

```

7.5.3 木と動的計画法

この節以降は8章のグラフの探索に馴染んでから取り組むことが適切。

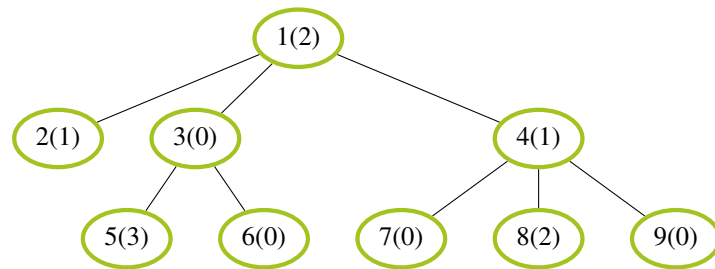


図 7.1 Sample 1

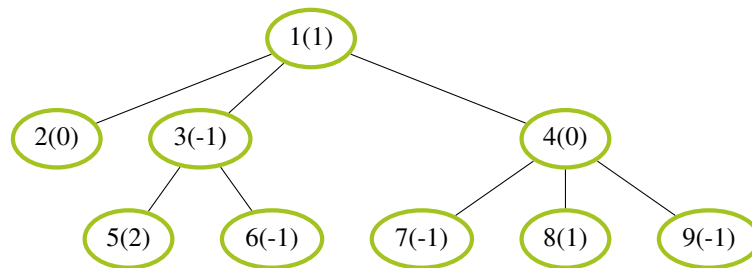


図 7.2 Sample 1(過不足)

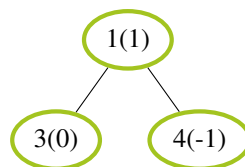


図 7.3 葉を調整 (2=0, 5=2, 6=-1, 7=-1, 8=1, 9=-1 を消去)

問題

TELE★

(Croatia OI 2002 Final Exam - Second Day)

(意識) 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>

ヒント: 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、 $(0,3), (1,2), \dots, (3,0)$ 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。

問題	Heap★★	(POJ Monthly–2007.04.01)
<p>ノードに整数が書かれた二分木がある。整数を書き換えて、どのノードに対しても、(左の子孫たち) $<$ (右の子孫たち) \leq (自分自身) を満たすようにしたい。最小何か所書き換えればよいか？</p> <p>木は完全二分木とは限らず、後ろのほうが詰まっていない場合がある。</p> <p>http://poj.org/problem?id=3214</p>		

入力形式注意

7.5.4 木の直径

問題	Fuel★	(Algorithmic Engagements 2011)
<p>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)</p> <p>https://szkopul.edu.pl/problemset/problem/5g0vDW-MvMGHfWQqh56jQKx1/site/</p>		

問題	Bridge Removal★	(国内予選 2014)
<p>群島の橋を渡りながら全ての橋を撤去しながら職人の、最短時間を求める。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1196&lang=jp</p>		

7.5.5 木の正規化

問題	部陪博士, あるいは, われわれはいかにして左右非対称になったか ★★	(国内予選 2007)
<p>式を表す二分木を与えられた規則で正規化せよ</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1152&lang=jp</p>		

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

7.5.6 木の中心

問題

Cave**

(11th Polish Olympiad in Informatics)

木の節点のどこかに宝が隠されている。質問に対する答えを元にその宝を見つける。質問回数を最小にするような質問戦略で必要になる、最大の質問回数を求めよ。

質問は、節点の一つを選んで、そこに宝があるかを聞く。宝がある場合はそこでゲームが終了し、そうでない場合は、どちら方向に宝があるかが隣接する節点により示される。

https://szkopul.edu.pl/problemset/problem/5Z9PRRPP-R90WhmbSY_qHd-1/site/

第 8 章

グラフ (2) グラフ上の探索

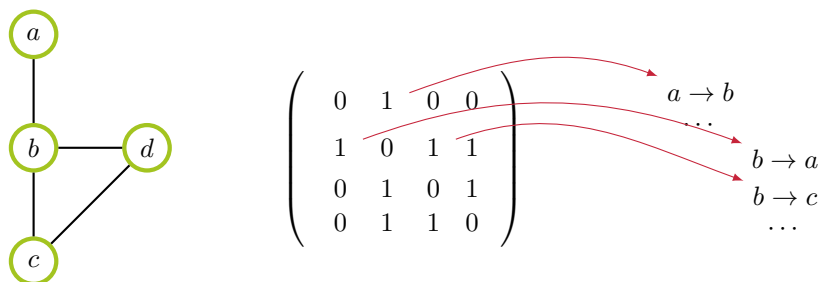
概要

連結なグラフの辺を全てたどる方法として、幅優先探索 (BFS) と深さ優先探索 (DFS) を紹介する。全部の辺をたどることから、グラフの連結性を判定したり、二部グラフかどうかを判定することが出来る。幅優先探索はさらに、根から各節点への距離を求めたり、木の直径を求めることができる。深さ優先探索は、トポロジカルソート、橋や間接点を求める、強連結成分分解などの応用がある。

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.

```
a → b
b → a c d
c → b d
d → b c
```

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^{攻略}[2, pp. 264–(12 章)], 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, 行毎に異なる)

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

この問題のように多くのケースでは, 頂点番号を $1, \dots, N$ 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, この資料では頂点番号から 1 を引いて, $0, \dots, N - 1$ と内部では扱い, 入出力の際に ± 1 して対応する. 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.

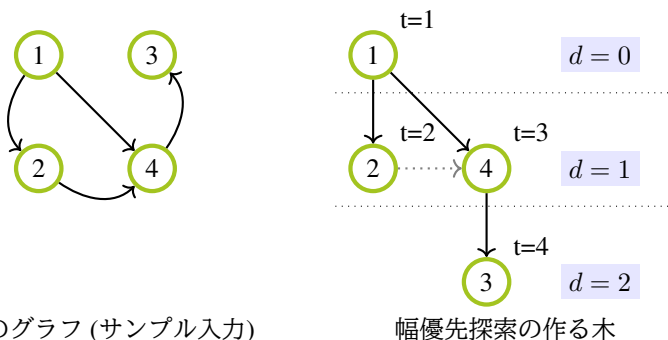
Python3

```
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)
<p>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)</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_11_C&lang=jp</p>		

幅優先探索 (参考書 ^{攻略}[2, pp. 282–], 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

問題のサンプル入力である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 (6.2.2 章参照) 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

```

BFS の過程で、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。10 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         }

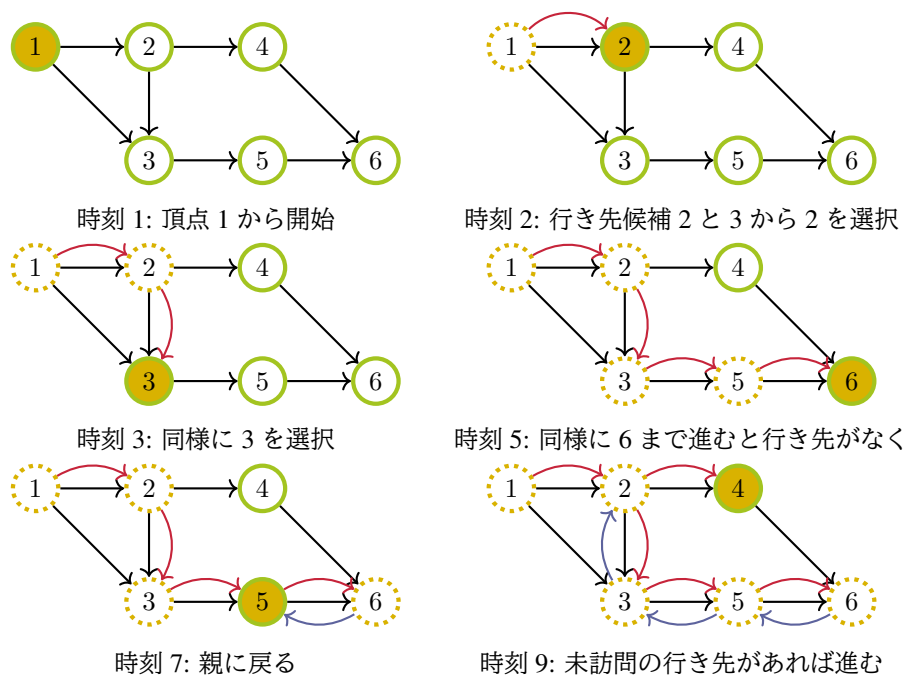
```

```
19     }
20 }
```

8.3 深さ優先探索 (DFS)

問題	Depth First Search	(AOJ)
<p>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&lang=jp</p>		

深さ優先探索 (参考書 ^{攻略}[2, pp. 273–], 参考書 [3, p. 33]) は、全ての節点を訪問する別の手法で、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。



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 }

```

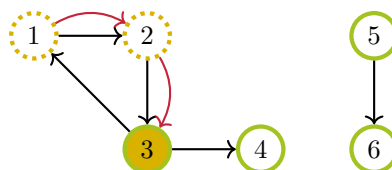
Python3

```

1 time = 1
2 D = [-1 for _ in range(N)]
3 F = [-1 for _ in range(N)]
4 def dfs(src):
5     global time # 関数内で global 変数の書き換えることを宣言
6     D[src] = time # 訪問時刻を記録
7     time += 1
8     for dst in range(N):
9         if ...: # src から dst に移動可能で, dst が未訪問だったら
10             dfs(dst)
11     F[src] = time # 訪問終了時刻を記録
12     time += 1
13     # 関数の終わりで親に戻る (青矢印)
14
15 for root in range(N): # 番号が若い節点から
16     if ...: # D[root] が未訪問だったら
17         dfs(root) # dfs を始める
18 # 出力

```

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る, この問題では上記のようなグラフも与えられる. 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, 見つければそこを出発点に DFS を行えば良い.

スタックを明示的に使った DFS (参考)

通常は、先に紹介した再帰を使った実装で十分である。但し、再帰の段数が深くなる場合はプロセスに割り当てられた stack 領域 (データ構造の stack とは区別せよ) が足りなくなり、segmentation fault などが起こることがある。実用上は、環境に応じた方法を用いて stack 領域の割り当てを増やせば十分であることが多いが、コンテストにおいてはそのような手段を用いることができない場合も多い。

データ構造のスタック (6.2.1 章) を用いる実装を次に示す。スタックに入れた (push した) データは、top() 及び pop() の操作により遅く入れた順に取り出される。主要な部分は、BFS の queue を stack に替えただけである (しかし queue と stack の違いから実際の探索の振る舞いは大きく異なる)。

```
C++
1 void dfs(int src) {
2     // cerr << "dfs root = " << src << endl;
3     stack<int> S;
4     S.push(src);
5     while (!S.empty()) {
6         int cur = S.top();
7         S.pop();
8         if (...) { // cur が初めての訪問の場合
9             // 初訪問時刻を記録
10            S.push(cur); // (*) 子孫の訪問を全て終えたらもう一度自分に戻る
11            for (...) { // 全ての子節点 dst について
12                // 兄弟に順番がある場合には、訪問したい順の逆順に for を設定する
13                // こと
14                if (...) { // cur から dst に辺があり、dst が未訪問なら
15                    S.push(dst); // todo 一覧に加える
16                }
17            }
18            else if (...) { // 今回が cur は二度目の訪問
19                // (*) に対応する子孫を巡り終わった後なので、cur の離脱時刻を記録
20            }
21            else if (...) { // 今回が cur への三度目以降の訪問
22                // 何もしない (後で訪問する予定だったが、
23                // 子孫節点経由で先に訪ねてしまったケース)
24            }
25        }
```

なお、節点を一度ずつ訪問すれば十分な場合は 10 行目の push は不要である。今回は訪問時刻だけでなく離脱時刻も必要なので、訪問と離脱でつごう二回ずつ各節点をスタックに入れている。また 11 行目の for 文は、問題で番号の若い節点から訪問することが求められている点とスタックは遅く入れた順に取り出される点を考慮して、順番を調整する必要がある。

8.4 連結性の判定

グラフが連結であるかは、BFS でも DFS でも、どちらでも求めることが出来る。

以下の問題では、問題文中でグラフの辺と頂点が明示的に与えられるわけではないが、自分でグラフを構成して探索を行うことで解を得ることが出来る。

問題

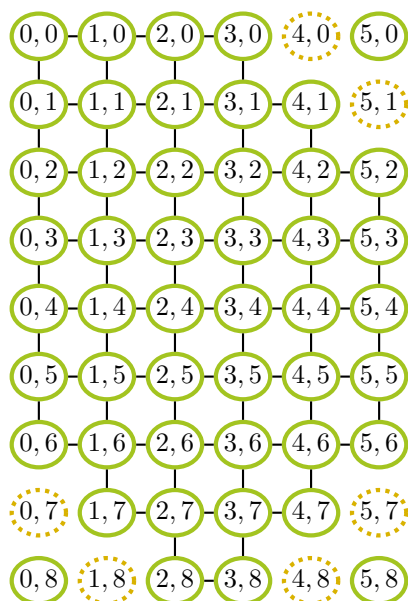
Red and Black

(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いくて、(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。

8.5 二部グラフの判別

各辺の両端の色が異なるように頂点を 2 色で (たとえば赤と青で) 塗り分けられるようなグラフを**二部グラフ**という。与えられたグラフが二部グラフかどうかを判定するには、BFS または DFS 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		

サンプル入力の解釈:



データの保持

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

入出力例:

```
C++
1 // 虫 id を id_color 色に割り当てて整合するかどうかを返す
2 // 整合する → true, しない → false
3 bool search(int id, int id_color) {
4     // ここを 3 種類作る
5 }
```

```
C++
1 int main() {
2     int scenarios;
```

```

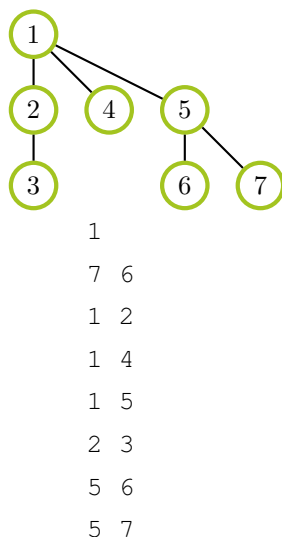
3      scanf("
4      for(int t=0; t<scenarios; t++){
5          //この for 文のブロックが一つの問題
6          fill(&e[0][0], &e[0][0]+2010*2010, 0); //eに初期化
7          fill(&color, &color+2010, 0); //colorに初期化
8          scanf("
9              for (int j=0; j<edges; ++j) {
10                  int src, dst;
11                  scanf("
12                  e[src][dst]=e[dst][src]=1; //両方向に通行化
13              }
14              bool ok=true;
15              for(int j=1; j<=bugs; ++j) //全ての虫について
16                  if(color[j]==0 && !search(j, 1)) {
17                      ok=false; //一回でも失敗したら、不整合
18                      break;
19                  }
20              if(t)
21                  printf("\n");
22              printf("Scenario #
23              if (! ok)
24                  printf("Suspicious_bugs_found!\n");
25              else
26                  printf("No_suspicious_bugs_found!\n");
27          }
28      }

```

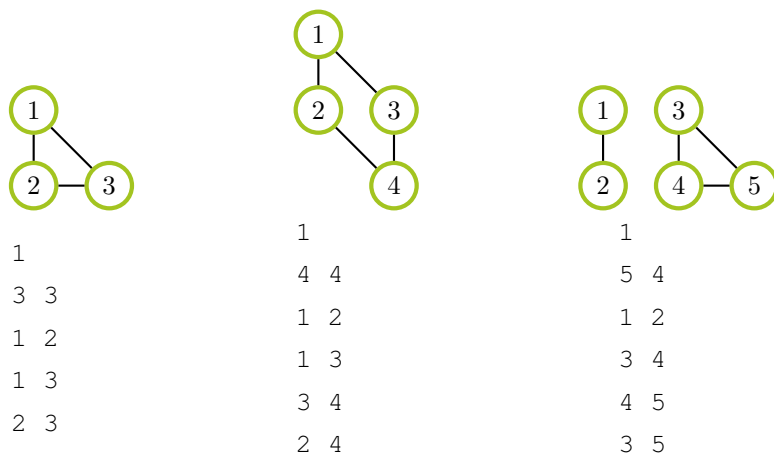
問題文中に注意が有る通り、poj の cin は、scanf の 10 倍以上遅いので、scanf を使う。

■グラフの走査 Bug's life を解くには、「全ての辺を通」って、各頂点を 1 と -1 で塗り分けられる (隣接する頂点の数字が異なる) ことを確認すれば良い。グラフで、「全ての辺を通」る方法である、幅優先探索 (BFS) と深さ優先探索 (DFS) でとくことができる。どちらでも良いが、ここでは DFS で作成してみる。

問題文の sample 入力は小さすぎるため、少し複雑なグラフで動作を確認する。



■合流/ループの検査 一般のグラフでは、ループを持つ場合がある。プログラムのとるべき挙動は、奇数のループと偶数のループで異なるので、二種類以上例題を作って確認する。



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

poj への提出の際は、動作確認用の cerr への出力は消しておくこと。

8.6 様々なグラフの探索

問題

Curling 2.0*

(国内予選 2006)

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

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

lang=jp

10 回までと限定があるので、DFS が向いている。DFS は、最大深さと兄弟節点数に比例するメモリを使用する。一方 BFS は、BFS 木で根からの距離が等しい節点数に比例するメモリを使用する。通常は後者のほうが大きい。

今回は、パックの位置の移動だけでなく壁が壊れるので、両方をモデル化する必要がある。

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		

取り除くとグラフが非連結になる頂点を関節点という。次にとりあげる、橋(取り除くとグラフが非連結になる辺)とともに、グラフの連結の強さを表す概念である。

関節点は DFS で求めることが出来る。グラフ G の深さ優先探索木を T とすると、関節点には以下の性質がある:

- T について根が複数の子を持つなら、関節点である。
- T のある頂点 n とその子 c について $n.depth \leq c.min$ が成り立つとき n は G の関節点である。
ただし $x.depth$ は、頂点 x の深さ (T での根からの距離) を表し、 $x.min$ は以下の最小値である:
 - $x.depth$
 - G で x に隣接するが T では隣接しない任意の頂点 y の $y.depth$,
 - T での x の任意の子孫 z の $z.min$

具体的には、 $x.depth$ や $x.min$ を判定しながら、深さ優先探索を行う。

問題	Bridges★	(AOJ)
橋を求めよ。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_3_B&lang=jp		

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

問題	Karakuri Doll★	(模擬国内予選 2007)
<p>からくり人形師の JAG 氏作の人形の動作を確認せよ。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2017&lang=jp</p>		

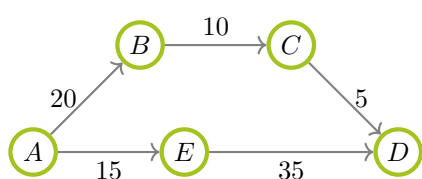
問題	Two Parties★★	(12th Polish Olympiad in Informatics)
<p>グラフを 2 分割して、全ての頂点の次数を偶数にできるか?</p> <p>https://szkopul.edu.pl/problemset/problem/eC-cABL-jWd4JdZDmfWufeeQ/site/</p>		

問題	Cakes★★	(Algorithmic Engagements 2009)
<p>菓子職人が沢山集まったので 3 人組みチームを作れるだけ作れ。チーム内の職人一人あたりが消費する小麦粉の量の最大値をチームとして消費して、チーム毎にケーキを作る。全チームの合計で最大どれだけの小麦粉を消費するか。一人の職人は複数チームに所属できるが、3 人でチームを組むにはチーム内のペア全ての関係が良好でなければならない。</p> <p>https://szkopul.edu.pl/problemset/problem/XWoCTR5RfPUYrlXIGPad9u2W/site/?key=statement</p>		

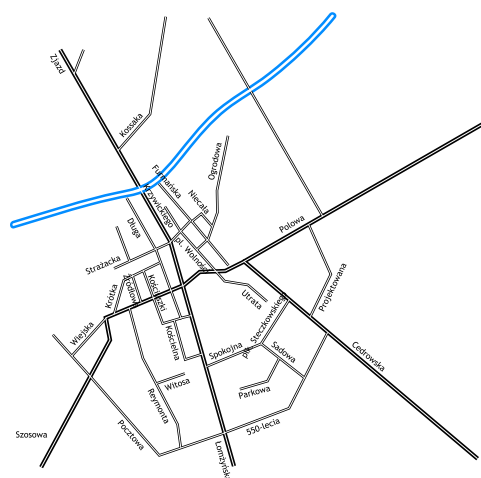
第9章

グラフ (3) 最短路問題

こんな問題



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

まずは, コストは非負であると仮定し (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), 後に一般の場合を考える. この章の前半でのグラフの表現としては, もっとも簡単な隣接行列を用いる (8.1, 参考書 [3, pp. 90, 91]). 隣接行列 K の i, j 要素 $K[i][j]$ は, i から j に有向辺があればそのコスト, ない場合は ∞ と表現する.

9.2 全点对間最短路

最短路問題を解くアルゴリズムには様々なものがあるが, 全ての節点間の最短路を求める Floyd-Warshall を, まず覚えてくとい (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  $K[i][j] = \infty$  (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
    ▷ 添字  $k$  を  $i$  や  $j$  と入れ替えると動かないので注意!
3:     for  $i = 1..N$  do
4:       for  $j = 1..N$  do
5:         if  $K[i][j] > K[i][k] + K[k][j]$  then
6:            $K[i][j] \leftarrow K[i][k] + K[k][j];$ 
    ▷  $k$  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 of 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 までを経由地を含む i から j の最短路 (の一つ) を D_{ij}^a と表記する。 $a \geq 2$ の時 D_{ij}^a 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_{ij}^{a-1} 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_{ia}^{a-1} と D_{aj}^{a-1} 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、 $D_{ij}^a = \min(D_{ij}^{a-1}, D_{ia}^{a-1} + D_{aj}^{a-1})$ となる。

9.2.1 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&lang=jp>

■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("
7      for(int i=0; i<M; i++){
8          scanf("
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いくても、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。

9.2.2 負の重みを持つ辺がある場合

例題

Shortest Path - All Pairs Shortest Path

(AOJ)

全頂点間の最短路を求めよ。ただし負の重みがありうる。

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_1_C&lang=jp

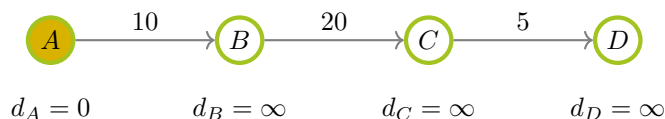
これまでは非負の重みを持つ辺のみを考えていたが、負の重みを持つ辺 (負辺) としてモデル化することが適切な場合もありうる。たとえば、ある区間では通行料を払う代わりに小遣いをもらえるスゴロクなどを考えよう。そのような場合に、最短路の概念はどう変化するだろうか? コストが負でありうる場合は非負の場合に成り立つ性質のいくつかは成り立たないため、注意が必要である。特に、コストの総和が負である閉路、負閉路 (negative weight cycle) がある場合には、そこを回り続けるとコストは下がり続けるために最短路が定義できない。始点から終点までの途上に負閉路が無ければ最短路は定義可能で、(正ではないかもしれないが) 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。たとえば上記の手順で 5 行目の、“if $K[i][j] > K[i][k] + K[k][j]$ ” という部分に、辺 ik と kj の存在を条件に加えること。辺のコストが非負であれば存在しない辺のコストを inf とすることで辺の存在をついでに確認できたが、負辺が存在する場合は片方が inf でも足すと小さくなることもある。

9.3 単一始点最短路

始点を一つ定めて始点から各点への最短路を求める問題は、全点对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。

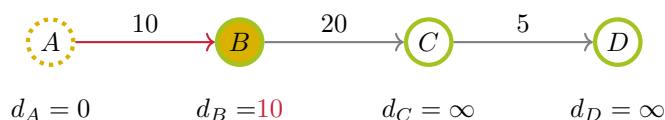
9.3.1 緩和 (relaxation)



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

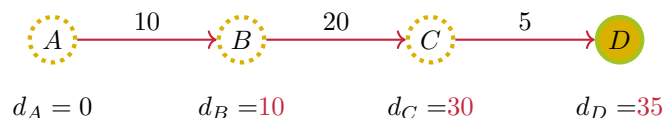
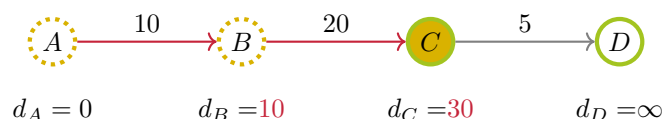
定義: $d[x]$ を A から x までの最短コストの上限とする。

初めに、 $d[A] = 0$ (A から A まではコストがかからない)、 $d[B] = d[C] = d[D] = \infty$ (情報が無いので ∞) 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。

9.3.2 Bellman-Ford 法

(参考書 [3, p. 95])

```

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

```

- 緩和の順番は? – 適当に通る
- 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)

9.3.3 Dijkstra 法

(参考書 [3, p. 96])

```

1: procedure DIJKSTRA( $V, E, w(u, v), v_s$ )
2:   for  $v \in V$  do
3:      $d[v] \leftarrow \infty$                                 ▷ 初期化: 始点から各頂点までの距離の上限は  $\infty$ 
4:      $d[v_s] \leftarrow 0$                                 ▷ 初期化: 始点から始点までの距離は 0
5:      $S \leftarrow \emptyset$                                 ▷ 最短距離が確定した頂点の集合, 最初は空
6:      $Q \leftarrow V$                                     ▷ 最短距離が未確定の頂点の集合, 最初は全て
7:     while  $Q \neq \emptyset$  do                            ▷ 最短距離が未確定の頂点なくなるまで繰り返す
8:       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
9:        $S \leftarrow S \cup \{u\}, Q \leftarrow Q \setminus \{u\}$     ▷  $u$  までの最小距離は確定
10:      for  $v \in Q$  s.t.  $(u, v) \in E$  do
11:         $d[v] \leftarrow \min(d[v], d[u] + w(u, v))$           ▷ 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)

9.3.4 手法の比較と負辺

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, $V = 100$ 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.

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。都市や辺の数が多いので、隣接行列ではなく隣接リストで表現すると良い。

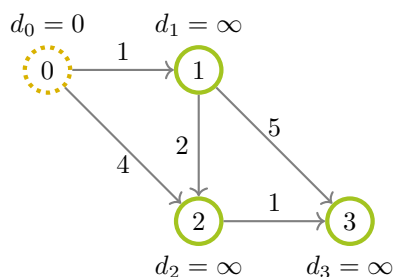
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_1_A&lang=jp



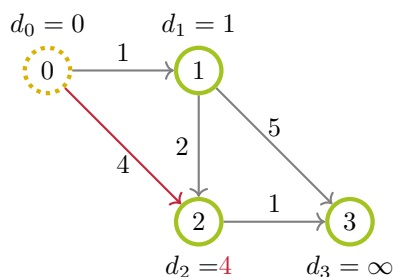
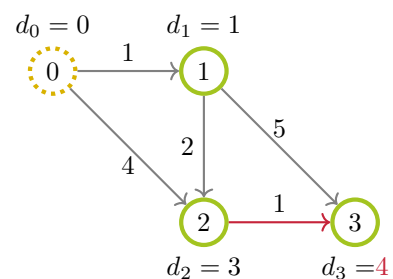
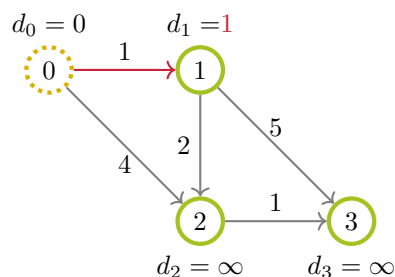
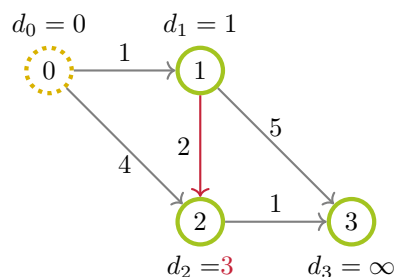
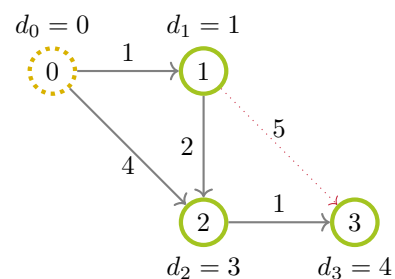
グラフの表現

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ず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 $\{0,2\}$ で緩和: $d_0 + 4 = 4 < \infty$ 辺 $\{2,3\}$ で緩和: $d_2 + 1 = 4 < \infty$ 辺 $\{1,3\}$ では更新が起こらない: $d_1 + 5 = 6 > 4$ 辺 $\{1,2\}$ で緩和: $d_1 + 2 = 3 < 4$ 辺 $\{1,2\}$ で緩和: $d_1 + 2 = 3 < 4$ 

C++

```

1  int V, E, R, S[500010], T[500010], D[500010]; // 問題で与えられる入力
2  int C[100010]; // 各頂点までの最短距離の上限
3  // 無限を表す定数を全頂点をたどる最大超に設定
4  const int Inf = 10000*100000+100;
5
6  int main() {
7      cin >> V >> E >> R;
8      for (int i=0; i<E; ++i)
9          cin >> S[i] >> T[i] >> D[i]; // 各辺を入力
10     ... // Cを初期化: C[R]を0に, 他をInfに
11     for (int t=0; t<V; ++t) { // V回繰り返す
12         bool update = false;
13         for (int i=0; i<E; ++i) {
14             int s = S[i], t = T[i], d = D[i]; // i番目の辺のs, t, dに対して
15             if (C[s] < Inf && ...) { // 辺s, tで緩和できるなら
16                 C[t] = ... // C[t]を更新;
17                 update = true; // 更新が起こったことを記録
18             }
19         }

```

```

20         if (!update) break; // 辺を一巡して更新がなければ打ち切って良い
21     }
22     ... // 出力
23 }

```

Python3

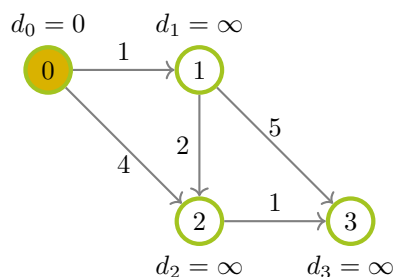
```

1 NV,NE,R = map(int,input().split()) # NXでXの個数を表した
2 Inf = 1001001001 # 全頂点を通るパスより大きな値
3 E = [tuple(map(int,input().split())) for _ in range(NE)]
4 # 初期化
5 D = [Inf for _ in range(NV)]
6 D[R] = 0
7 # メインのループ
8 for t in range(NV):
9     update = 0
10    for s,t,d in E:
11        # try to decrease D[t] w.r.t. edge (s,t) with cost d, here
12        # increment update if D[t] was changed (decreased)
13        if update == 0:
14            break
15 for v in range(NV):
16     print(D[v] if D[v] != Inf else "INF")

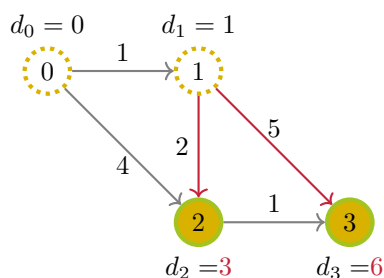
```

■Dijkstra 法で解いてみよう Dijkstra 法 (9.3.3 節) で解く場合は、優先度つきキュー (**priority queue**, 6.2.3 章参照) を使うと便利である。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。

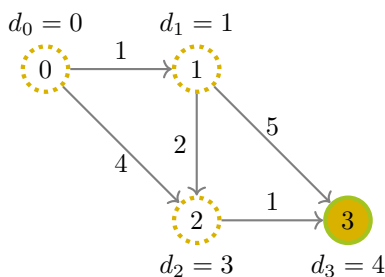
上記の方針で実装した Dijkstra 法の動作例を、以下に示す。優先度つきキュー P は、〈始点からの距離, 頂点番号〉のペアを要素として持ち、距離の小さい順に、先頭要素 (左端) が取り出されるとする。図のグラフ 1 つが手順 7 行目からのループの 1 回の実行に対応する。



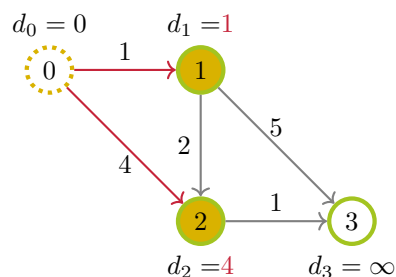
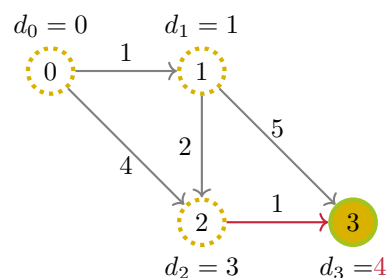
初期状態

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

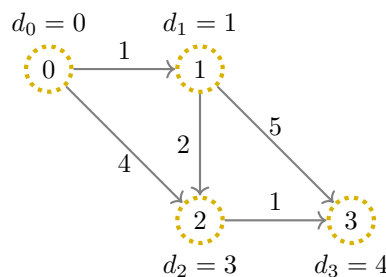
頂点 0 を確定して頂点 1, 2 を緩和

 $P = (\langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 3 \rangle)$ 

頂点 1 を確定して頂点 2, 3 を緩和

 $P = (\langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 6 \rangle)$ 頂点 2 は既に確定しているため $\langle 4, 2 \rangle$ は無視 $P = (\langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 3 \rangle)$  $\langle 0, 0 \rangle$ 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

 $P = (\langle 1, 1 \rangle, \langle 4, 2 \rangle, \langle 3, 4 \rangle)$  $\langle 4, 2 \rangle$ を取り出し頂点 3 が距離 4 で確定 $P = (\langle 3, 4 \rangle)$

問題

Single Source Shortest Path II

(AOJ)

重みが負の辺がある場合に、都市 0 から全ての都市への最短路を求めよ。詳しくは問題文を参照。

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_1_B&lang=ja

重みが負の辺がある場合は、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。

問題	Wormholes	(USACO 2006 December Gold)
<p>ワームホールを通じて過去に戻れるルートを探す。出発地はどこを選んでも良いので、その場所の過去の時刻に戻れば良い。</p> <p>http://poj.org/problem?id=3259</p>		

9.4 練習問題

問題	Railway Connection★	(国内予選 2012)
<p>最も安い運賃の経路を求める</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1182&lang=jp</p>		

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★	(国内予選 2007)
<p>崖を登る最短の時間を求める</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1150&lang=jp</p>		

問題	Magical Dungeon★	(冬合宿 2008)
<p>ヒットポイント H を持つ勇者が部屋 S から移動して部屋 T に到着した瞬間にモンスターと戦う。最大いくつのヒットポイントで戦闘を迎えられるか? 通路には正負の数が割り当てられていて、正ならヒットポイントを回復し、負なら失う。ヒットポイントが 0 以下になるような移動はできない。ヒットポイントが H を超えるような移動は可能ではあるが、ヒットポイントは最大 H までしか回復しない。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2124&lang=jp</p> <p>問題とデータセット: http://acm-icpc.aitea.net/index.php?2007%2FPractice%2F%E5%86%AC%E5%90%88%E5%AE%BF%2F%E5%95%8F%E9%A1%8C%E6%</p>		

	(問題 続き)
<p>96%87%E3%81%A8%E3%83%87%E3%83%BC%E3%82%BF%E3%82%BB%E3%83%83%E3%83%88 (day3, C)</p>	

注: 負の閉路を

少し工夫が必要.

問題	壊れたドア ★	(国内予選 2011)
<p>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.</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1178&lang=jp</p>		

問題	The Most Powerful Spell★★	(国内予選 2010)
<p>作成可能な中で, 辞書順で最も早い呪文を求める.</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1169&lang=jp</p>		

問題	Sums★	(10th Polish Olympiad in Informatics)
<p>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)</p> <p>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.</p> <p>https://szkopul.edu.pl/problemset/problem/4CirgBfxbj9tIAS2C7DWCCd7/site/</p>		

第Ⅱ部

トピックス

第 10 章

平面の幾何

10.1 概要: 点の表現と演算

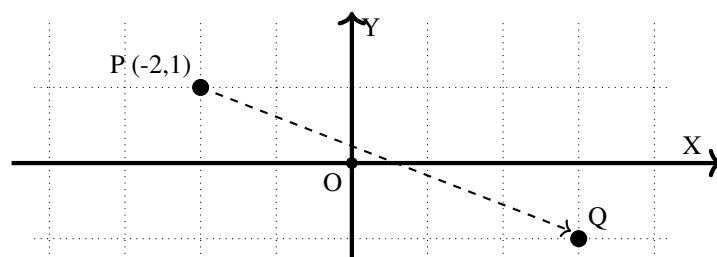
概要

さまざまな状況で計算機で図形的問題を扱う機会に出会う。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する場合がある。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。

平面上の点を C や C++ で表現する場合には, たとえば構造体を用いる方法がある (参考書 ^{攻略}[2, pp. 365–(16 章)]。この資料では, 少し楽をして以下のように複素数で表現する

10.1.1 複素数による点の表現

■C++ の複素数による表現 C++ では標準ライブラリの `complex` 型を用いる。(実部 `real()` を x に, 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), Q; // 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  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
9  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
10 cout << abs(P) << endl; // ベクトル OP の長さ
11 cout << norm(P) << endl; // norm(P) = abs(P)2

```

■C の複素数による表現 この資料では、C の使用は非推奨であるが、後述する注意事項のために、一応掲載する。

C (gcc または C99) 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("
6  a *= b; // 乗算
7  printf("

```

⚠ 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。知らないと思つけない、と思われる。

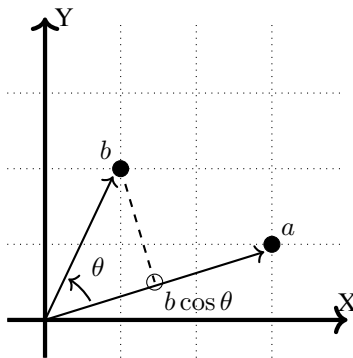
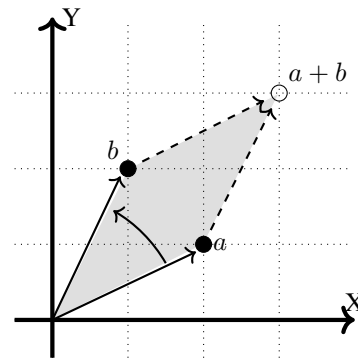
■Python の複素数による表現 Python3 においても複素数 complex をほぼ同様に利用可能である。詳細は help(complex) や help(cmath) で確認のこと。

```

Python3
1  import cmath
2  import math
3  P = complex(-2, 1) # (-2+1j) も可
4  print(P)
5  print(P.real)      # 実部
6  print(P.imag)      # 虚部
7  Q = P + complex(5, -2) # 平行移動
8  Q *= complex(math.cos(a), math.sin(a)) # 回転
9  print(abs(P))      # 長さ

```

10.1.2 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:  $a.x*b.x + a.y*b.y$ 
2 double dot_product(xy_t a, xy_t b) { return (conj(a)*b).real(); }
3 // 図 (2) クロス (外) 積, ベクトル  $a, b$  が作る三角形の符号付き面積の二倍:  $a.x*b.y - b.x*a.y$ 
4 double cross_product(xy_t a, xy_t b) { return (conj(a)*b).imag(); }
5 // (対応図なし) 投影 原点と  $b$  を結ぶ直線に点  $p$  を投影
6 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、直線上に点を投影する際に便利である。

Python3

```

1 def norm(c):
2     a = abs(c)
3     return a*a
4 def dot_product(a, b):
5     return (a.conjugate()*b).real
6 def cross_product(a,b):
7     return (a.conjugate()*b).imag
8 def projection(p, b):
9     return b*dot_product(p,b)/norm(b)
```

10.2 三角形の符号付き面積の利用

10.2.1 多角形の面積

例題

Area of Polygon

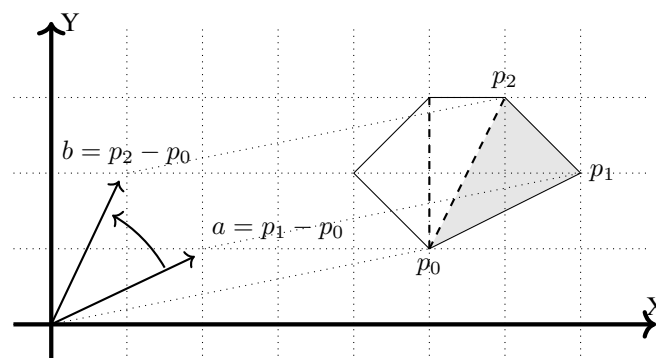
(PC 甲子園 2005)

凸多角形の面積を求めよ。

注 1: 問題文中にはヘロンの公式が書いてあるが, 下記の符号付き三角形で解くこと。(あとで凸でない多角形の面積に応用するため)

注 2: 頂点列は順に与えられるが, 時計回りか反時計回りかは指定がないため, 最後に絶対値をとる。

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



(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("
7      %lf%lf", &x, &y)) 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[i], b=P[i+1], c=P[i+2];
13         sum += a.b.c; // 三角形 abc の面積を加算
14     }
15     printf("

```


16 }



scanf で入力が続く限り読む方法

入力行を読める限り読むために、サンプルコード中では `while (~scanf("%lf,%lf", &x, &y))` というループを採用した。これは、短い(しかし汎用性の低い)書き方で、`~EOF` が 0 になる環境でのみ使用可能である。

Python3

```

1 P = [] # 頂点列
2 try:
3     while True:
4         x,y = map(float,input().split(',') )
5         P.append(complex(x,y))
6 except EOFError:
7     pass
8 N = len(P) # 頂点の個数
9 total = 0.0
10 for i in range(1,N-1):
11     a,b,c = P[0],P[i],P[i+1]
12     total += .. // 三角形 abc の面積を加算
13 print("

```



Python3 で入力が続く限り読む方法

`input()` を試みて入力が終わっていた場合、Python では `EOFError` という例外が発生するので、あらかじめ `try` で囲み `except` で検知する。例外が発生すると `input()` の書かれていた `while` ループを脱出して、外側の `except` ブロックが実行される(この場合は `pass` で、何もせずに次の行に移る)。

例題

Polygon - Area

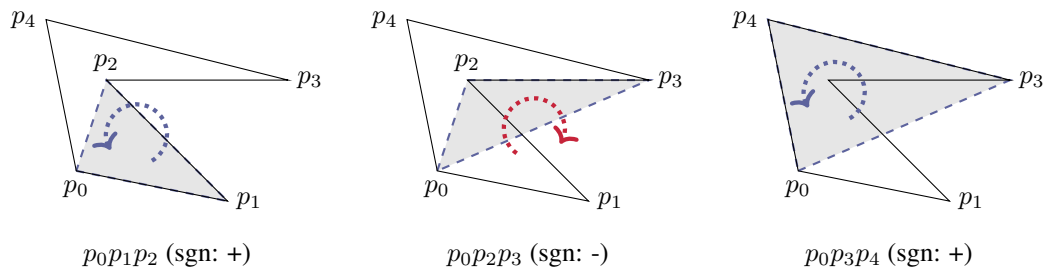
(AOJ)

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&lang=jp> 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。



10.2.2 平行の判定

例題

Parallelism

(PC 甲子園 2003)

概要: $A = (x_1, y_1)$, $B = (x_2, y_2)$, $C = (x_3, y_3)$, $D = (x_4, y_4)$ の異なる 4 つの座標点を与えられたとき、直線 AB と CD が平行かどうかを判定せよ。

点の座標は小数点以下最大 5 桁までの数字を含む実数で与えられる (この情報を数値誤差の推定に用いる)。

<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 展開が用いられる。^{*1}

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

Python3

```

1 N = int(input())
2 for _ in range(N):
3     P = list(map(float, input().split()))
4     a, b, c, d = [complex(P[i*2], P[i*2+1]) for i in range(4)]
5     parallel = ... # abとcdが並行かを表す真偽値を計算
6     print("YES" if parallel else "NO")

```

🔥 動作確認

この問題はサンプル入出力例が少なく、またジャッジデータも非公開である。そのため、自分でテストデータを試すと良い。たとえば、長い線、短い線、面積の符号の正負などが、確認すべき例である。

たとえば、以下の例は全て “NO” である。

```

-0.00001 0 0.00001 0 -0.0001 0 0.00001 0.00001      1
-100 100 100 100 -100 100 100 99.99999                2

```

■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 (B.4 章も参照)。この問題での入力値は、絶対値が 100 以下かつ各値は小数点以下最大 5 桁までの数値と明示されているので、 10^5 倍して整数 (long long) で扱えば誤差の影響を避けることができる。あるいは、サンプルコードの eps のように、誤差の範囲を予測する方法もある。二つのベクトル (a, b) と (c, d) にそれぞれの要素に誤差が加わった時に、(1) 平行の場合に $|ad - bc|$ の取る最大値 (誤差がなければ 0) と、(2) 平行でない場合に $|ad - bc|$ の取る最小値を比較して、(1) < 閾値 < (2) となるよう閾値をとる。粗く見積もると (1) は最大 $(4 \cdot 100) \cdot (100 \cdot 2^{-54}) \approx 2.2 \cdot 10^{-12}$ 程度 ($100 \cdot 2^{-54}$ は 100 までの数を double で表した時の表現誤差、400 は $|(a + \varepsilon)(d + \varepsilon) - (b + \varepsilon)(c + \varepsilon)|$ 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。

10.2.3 内外判定

例題

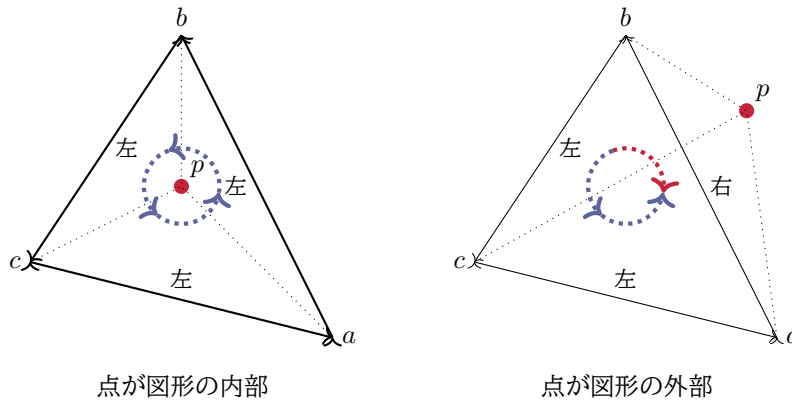
A Point in a Triangle

(PC 甲子園 2003)

平面上に $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ を頂点とした三角形と点 $P(x_p, y_p)$ 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  int main() {
3      while (true) {
4          for (int i=0; i<4; ++i) cin >> x[i] >> y[i];
5          if (!cin) break;
6          xy_t a(x[0],y[0]), b(x[1],y[1]), c(x[2],y[2]), p(x[3],y[3]);
7          // pab の符号付き面積の 2 倍は, cross-product (a-p,b-p)
8          // pbc の符号付き面積の 2 倍は, cross-product (b-p,c-p)
9          // pca の符号付き面積の 2 倍は, cross-product (c-p,a-p)
10         bool ok = 符号が揃っている
11         cout << (ok ? "YES" : "NO") << endl;
12     }
13 }

```

凸とは限らない多角形の内外判定については、次の節を参照。

10.2.4 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。

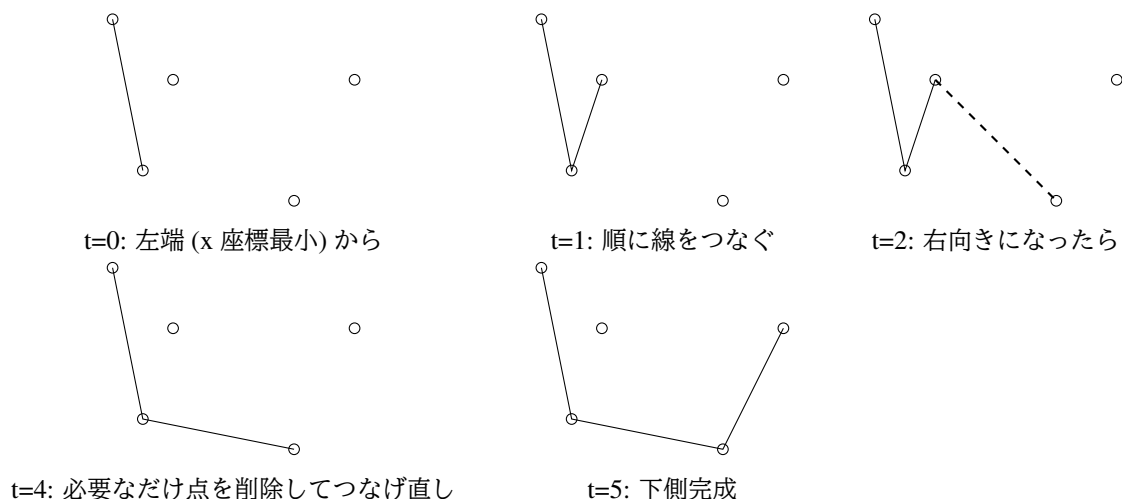
(注: 辺上の点を出力させる, 少し特殊な設定である)

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&lang=jp> 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) から順に右に向かい, まずは下半分の外周を構成する。途中現在の進行方向より右に向かうことになったら, (本来凸法に含まれない点を入れてしまった状況であるので) 原因となった点をすべて取り除く。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 $O(N \log N)$ 。



■点の整列 C++ の複素数型 (complex) には、比較演算子が定義されていないので、自分で定義する必要がある。下記の例のように std ネームスペースで operator< を定義すると、sort で自動で使われる。比較基準としては、 $!(a < b)$ かつ $!(b < a)$ の場合に $a == b$ であるようなものを用いること。^{*2}

```
C++
1 namespace std {
2     bool operator<(xy_t l, xy_t r) {
3         return (l.real() != r.real()) ? l.real() < r.real() : l.imag() < r.imag();
4     }
5     // 別案: std::pair と揃える場合
6     bool operator<(xy_t l, xy_t r) {
7         return make_pair(l.real(), l.imag()) < make_pair(r.real(), r.imag());
8     }
9 }
```

10.3 様々な話題

例題

A Symmetric Point

(PC 甲子園 2005)

点 Q と線対称の位置にある点を入力せよ。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0081&lang=jp>

回答例: 直線上に Q を投影した点を S とする。すると求める点 R は、S をベクトル QS だけ平行移動した位置にある。カンマ区切りで与えられる入力の処理と、また小数点以下の出力桁数の制御には、`#include<cstdio>` して以下のように `scanf` と `printf` を用いるのが簡便である。

C++

^{*2} https://en.cppreference.com/w/cpp/named_req/Compare

```

1  #include <stdio>
2  double X1,Y1,X2,Y2,XQ,YQ;
3
4  int main() {
5      while (~scanf("
6      _____&X1,_%Y1,_%X2,_%Y2,_%XQ,_%YQ) ){
7          xy_t_P1(X1,Y1),_P2(X2,Y2),_Q(XQ,YQ);
8          xy_t_R=_...; //_線対称の点を計算
9          printf("
10     }
11 }
```

問題

Polygon - Polygon-Point Containment

(AOJ)

凸とは限らない多角形について点の内外を判定せよ。

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_3_C&lang=jp

(注: 通常は点が辺上かどうかを判定することは難しいが、今回は整数座標なので可能)

回答例: 調べたい点から任意の方向に半直線を伸ばし、横切る辺の数を数える。偶数ならば外部、奇数ならば内部である。線分が直線と交点を持つかどうかなどの関数を事前に用意しておく必要がある。半直線が頂点のすぐ近くを通過する場合は、誤差の影響を心配しないで済むよう、角度を変えたほうが無難。

問題

Point Set - Closest Pair*

(AOJ)

最近点对を求めよ。

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=CGL_5_A&lang=jp

点の個数を N して、点のペアを全て試すと $O(N^2)$ の時間が必要だが、分割統治により $O(N \log N)$ で可能。なお $O(N)$ の乱択アルゴリズムも存在する。

分割統治の方針: X 座標毎と Y 座標毎のそれぞれで点をソートしておく。 X 座標で点を半分に分け、左と右でそれぞれ最近点对を再帰的に求める。求めた二つの最小距離の小さい方を d とする。全体の最近点对は、左のみの最近点对、右のみの最近点对、左の点と右の点の最近点对のいずれかである。後者は左右の区切りの線から距離 d 以内のものについて Y 座標順に並べた時に、定数 (たとえば 8) 以内のペアのみ候補となる性質を用いると、点の個数に対して線形の計算ステップで判定可能である。証明は区切りの線の周囲の適当な正方形グリッドを考えると、 d の制限で点があまり密に存在できないことから。なお、実装では、毎回 Y 座標順にソートしていると $O(N \log N)$ を実現できない。初めに一度全体をソートしておき、分割の際にそこから振り分けると良い。また分割の際には、複数の点と同じ Y 座標を持つ場合に注意を払う必要がある。実用的には、初めに全体をランダムに回転させると避けられる。

10.4 応用問題

問題	ConvexCut	(夏合宿 2012)
<p>与えられた図形に、どの角度で分割しても同じ面積で切れるような点が存在するかを求める。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2442&lang=jp</p>		

多角形の頂点が偶数であり、かつ組みになるべき辺が全て平行である時に限られること、あるいは組になる頂点の中点が等しいことなどで判定できることが証明できる。

回答例 (入出力)

```

C++
1  #include <complex>
2  #include <iostream>
3  #include <cstdio>
4  using namespace std;
5  int N, x, y;
6  typedef complex<double> xy_t;
7  xy_t P[60];
8  void solve() {
9      ...
10 }
11 int main() {
12     while (cin >> N) {
13         for (int i=0; i<N; ++i) {
14             cin >> x >> y;
15             P[i] = xy_t(x,y);
16         }
17         solve();
18     }
19 }

```

回答例 (中点の計算)

```

C++
1  void solve() {
2      // 奇数だったらそのような点はない
3      ...
4      xy_t a = (P[0]+P[N/2])*0.5; // P[0] と P[N/2] の中点
5      for (int i=1; i<N/2; ++i) {
6          xy_t b = ... // P[i] と P[i+N/2] の中点
7          // aとbが誤差を加味しても一致していなければ, abs(a-b) > eps, 求める点はない
8      }
9      printf("
10 }

```

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、それらの点になるべくたくさん囲むようにする。このとき、最大でいくつの点を同時に囲めるかを答えなさい。ここで、ある円が点を「囲む」とは、その点が円の内部または円周上にあるときをいう。(この問題文には誤差に関する十分な記述がある)

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

候補となる円の位置は無数にあるので、候補を絞る。(「ぎりぎりを考えよ」参考書 [3, p. 229])



考え方

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

n 個を囲んだ円があり、一点が円上に乗っているとする。その点を

しかない。



例外ケースに注意

上記の考え方は概ね正しいが、例外ケースがある。すなわち答えが
記の性質を持たない。

上



点の数え方

ある点 p にピッタリ載る円の位置を設定し、それに含まれる点の個数を調べたいとする。そのためには全ての点について円の中心との距離を計算すれば、概ね判定可能である。しかし、点 p そのものについては、円上に位置するため、数値誤差により外と判定されてしまうかもしれない。ここで許容誤差を大きくすると、本当に外にある点を内と判定してしまうリスクがある。そのため点 p の判定は特別扱いし、

回答例:

C++

```
1 int main() {
2     // 最大値を初期化
3     for (/*点p*/ ) {
4         for (/*点q*/ ) { // p!=q
5             if (/*pqを通る円があれば*/ ) { // 0個, 1個, 2個の場合がある
6                 /*全ての点をかめながら、内部の個数を数える*/
```



```

7          /*最大値を越えていれば更新する*/
8      }
9  }
10 }
11 }

```

問題	Roll-A-Big-Ball	(国内予選 2008)
条件を満たす最大の大玉の大きさを求める. http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1157&lang=jp		

問題	Space Golf	(アジア大会 2014)
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1348&lang=jp		

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&lang=jp		

問題	Area of Polygons★	(アジア大会 2003)
<p>網掛け部分の面積を求める。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1242&lang=jp</p>		

考え方: 薄切りにして和を求めるだけなのだが, 複数の線が同じますを通る場合など, 注意点がそれなりにある。

参考: <http://www.ipsj.or.jp/07editj/promenade/4501.pdf>

問題	Treasure Hunt★	(夏合宿 2012)
<p>領域内の宝を (効率的に) 数える</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2426&lang=jp</p>		

純朴に数えていると時間がかかりすぎるので, 点を与えられた時点で前処理を行い, 質問に答える準備を整えておく。質問に効率的に答えるためのデータ構造としては, たとえば四分木 (**quad tree**) を使うことができる。

問題	Altars★★	(6th Polish Olympiad in Informatics)
<p>中国では悪霊は直線上に進むと信じられているという枕。</p> <p>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。</p> <p>http://main.edu.pl/en/archive/oi/6/olt</p>		

問題	Fish★★	(Algorithmic Engagements 2009)
<p>毎日同じ時刻に起床/就寝する魚がいる。寝て起きると, 海流の影響で一マスずれていることがある。昨日の同時刻の自分の位置が見えるように泳ぐ。加減速は自在。一日の魚の動きを観察したルートの記録がたくさんあるが, 日付が分からない。最小で何尾の魚を観察していると言えるか。</p> <p>archipelago=群島</p> <p>http://main.edu.pl/en/archive/pa/2009/ryb</p>		

第 11 章

簡単な構文解析

こんな問題

特定の文法で書かれた文字列を計算機に理解させよう:

- $(V|V) \& F \& (F|V) \rightarrow F$ (真偽値の計算)
- $35=1?((2*(3*4))+(5+6)) \rightarrow '+'$ (演算子の推定)
- $4*x+2=19 \rightarrow x=4.25$ (方程式を解く)
- $C_2H_5OH+3O_2+3(SiO_2) == 2CO_2+3H_2O+3SiO_2$ (分子量の計算)

11.1 四則演算の作成

11.1.1 足し算を作ってみよう

■大域変数 :

```
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 }
```

```
Python3 1 S = "0"
        2 cur = 0
        3
        4 a = parse()
        5 assert a == 15
        6 assert cur == len(S)
```

■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 // 1文字読むが cur を進めない
   10 char peek() {
   11     assert(cur < S.size());
   12     return S[cur];
   13 }

```

Python で、関数内でグローバル変数を変更するには、global で指定する。

```

Python3 1 def readchar():
    2     global cur
    3     c = S[cur]
    4     cur += 1
    5     return c
    6 def peek():
    7     return S[cur]

```

■assert って何? (再掲)

```

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

```

最初の足し算

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)

- $P := Q \rightarrow P$ という名前の文法規則の定義
- $A B \rightarrow A$ の後に B が続く
- $'a' \rightarrow \text{文字 } a$
- $x \mid y \rightarrow x \text{ または } y$

文法通りに実装する (Digit)

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

C++

```
1 #include <cctype>
2 int digit() {
3     assert(isdigit(peek())); // S[cur] が数字であることを確認
4     int n = readchar() - '0'; // '0' を 0 に変換
5     return n;
6 }
```

文字の数字への変換: C や C++ では文字はその文字を表す数字で管理される。言語の規格では文字コードが規定されていないが、現状で我々が使う環境では ASCII コードと考えて良い。その中で、'0','1','2',..., 'a','b','c' などの文字の順番でコードが割り当てられていることを利用すると、上記の減算により '0' からいくつ後の文字かが分かり、それがすなわち求めたい数値である。ASCII コード表を確認する手段の一つは、man コマンドが簡便で、ターミナルで `man ascii` とすると良い。

Python3

```
1 def digit():
2     assert peek().isdigit()
3     n = int(readchar()) # 1 文字読んで数値に変換
4     return n;
```

Python の場合は、上記のように `isdigit()` で判定し、`int` で変換すると良い。

文法通りに実装する (Number)

`Number := Digit の繰り返し`

C++

```
1 int number() {
2     int n = digit();
3     while (cur < S.size() && isdigit(peek())) // 次も数字か 1 文字先読
4         n = n*10 + digit();
5     return n;
6 }
```

Python3

```
1 def number():
2     n = digit()
3     while cur < len(S) and peek().isdigit():
4         n = n*10+digit()
```

```
5      return n
```

文法通りに実装する (Expression)

Expression := Number '+' Number

```
C++ 1 int expression() {
    2     int a = number();
    3     char op = readchar();
    4     int b = number();
    5     assert(op == '+');
    6     return a + b;
    7 }
```

足し算だけならこれで動くはずである:

```
C++ 1 const string S = "12+3";
    2 size_t cur = 0; // 解析開始位置
    3 ..
    4 int parse() { return expression(); }
    5 int main() {
    6     int a = parse();
    7     cout << a << endl; // 15 が出力されるはず;
    8 }
```

■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 (cur < S.size() && (peek() == '+' || peek() == '-')) {
    4         // 足し算か引き算が続く間
    5         char op = readchar();
```

```

6         int b = number();
7         if (op == '+') sum にbをたす;
8         else sum からbを引く;
9     }
10    return sum;
11 }
```

次の拡張

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

11.1.2 カッコを使わない四則演算の (いい加減な) 文法

四則演算の (いい加減な) 文法

```

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

読みかた: (参考: (Extended) BNF)

- $A B \rightarrow A$ の後に B が続く
- $\{C\} \rightarrow C$ 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 }
```

Python では、// 演算子で整数除算が可能だが、この問題の仕様とは少し異なる。すなわち、この問題では $3/-2$ を -2 ではなく -1 と評価する必要があるようである。そこで、`math.trunc` 関数を用いる。

Python3

```
1 import math
2 def term():
3     a = number()
4     while cur < len(S) and (peek() == '*' or peek() == '/'):
5         op = readchar()
6         b = number()
7         a = a*b if op == '*' else math.trunc(a/b)
8     return a
```

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 }
```

11.1.3 四則演算: カッコの導入

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

カッコを導入した文法

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

`factor()` の実装例は以下:

C++

```
1 int expression(); // 前方宣言
2 int factor() {
3     if (peek() != '(') return number();
4     readchar(); // '(' を読み捨てる
5     int n = expression();
6     assert(peek() == ')');
7     readchar(); // ')' を読み捨てる
8     return n;
```


9 }

`term()` の実装も、文法に合わせて調整すること。

11.1.4 まとめ

実装のまとめ:

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

文法の記述:

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

■補足 $P := A \{ '+' A \}$ の繰り返しを再帰で記述する?

- $P := P '+' A \mid A$
→ 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
 - $P := A P'$
 - $P' := '+' A P' \mid \epsilon$
- (ϵ は空文字列)

11.2 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&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

Molecular Formula

(アジア地区予選 2003)

各原子の原子量を元に、各分子の分子量を求める。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1244&lang=jp>

初めに表を作成して、原子を原子量に変換できるようにしておけば、後は乗算と加算の演算。

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. 変数があ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 = S[cur++];
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 = S[cur++];
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

Matrix Calculator

(アジア地区予選 2010)

行列の掛け算をしよう

	(問題 続き)
http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1314&lang=jp	

問題	ASCII Expression	(アジア地区予選 2011)
ASCII art 風に描かれた算数を計算する. http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1322&lang=jp		

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&lang=jp		

回答例: 各分子の構成を調べて, 連立方程式を解く.

問題	Questions**	(Algorithmic Engagements 2008)
P 人の王子と魔法使いのそれぞれの知識状態を上手にシミュレートして, 質問になんと答えるかを当てる. http://main.edu.pl/en/archive/pa/2008/pyt		

補足

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

第 12 章

繰り返し二乗法と行列の冪乗

概要

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

応用先:

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

(参考書 [3, p. 114])

12.1 考え方

例として $3^8 = 6561$ の計算を考える．

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

練習: 3^{128} の場合は? $\rightarrow \log(128)$ 回の乗算

例題

Elementary Number Theory - Power

(AOJ)

2つの整数 m, n について、 m^n を 1000 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

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: $(a*b)\%M = ((a\%M)*(b\%M))\%M$

参考書 攻略[2, pp. 445–] 参照.

12.2 言語機能: struct と再帰関数

この章では long long と struct を使う. 必要に応じて付録 B.3 章と B.5 章を参照のこと.

12.3 正方行列の表現と演算

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

```
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
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);
```

12.4 練習: フィボナッチ数

問題	Fibonacci	(Stanford Local 2006)
<p>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.</p> <p>$0 \leq n \leq 10^{16}$</p> <p>http://poj.org/problem?id=3070</p>		

12.4.1 様々な計算方針

方針 1: 定義通りに計算する

```
C++
1 int fib(int n) {
2     if (n == 0) return 0;
3     if (n == 1) return 1;
4     return fib(n-2)+fib(n-1);
5 }
```

使用例:

```
C++
1 int main() {
2     for (int i=1; i<1000; ++i)
3         cout << "fib" << i << " = " << fib(i) << '\n';
4 }
```

$n = 30$ くらいで、先に進まなくなる。

方針 2: 一度行なった計算を記憶する

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

```
C++
1 int table[2000]; // 2000 まで答えを記憶
2 int fibmemo(int n) {
3     if (n == 0) return 0;
4     if (n == 1) return 1;
5     if (table[n] == 0) // もし初見なら
6         table[n] = fibmemo(n-2)+fibmemo(n-1); // 計算して覚える
7     return table[n]; // 覚えていた値を返す
8 }
```

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

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

方針 3: 小さい方から計算する

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

```
C++
1 int fibl(int n) {
2     int a[2] = {0,1};
3     for (int i=2; i<=n; ++i) {
4         // 不変条件: ループ開始時に、a[0] と a[1] はそれぞれ
5         // Fib(i-1) と Fib(i-2) (i が奇数)
6         // Fib(i-2) と Fib(i-1) (i が偶数)
7         // に相当
8         a[i]
9     }
10    return a[n]
11 }
```

12.4.2 行列で表現する

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

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

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

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

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

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.

12.5 応用問題

問題	One-Dimensional Cellular Automaton	(アジア地区予選 2012)
(意訳) 行列を T 乗する ($0 \leq T \leq 10^9$) http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1327&lang=jp		

$N \times N$ の行列の実装例:

```
C++
1 struct Matrix {
2     valarray<int> a;
3     Matrix() : a(N*N) { a=0; }
4 };
5 Matrix multiply(const Matrix& A, const Matrix& B) {
6     Matrix C;
7     for (int i=0; i<N; ++i)
8         for (int j=0; j<N; ++j)
9             C.a[i*N+j] = (A.a[slice(i*N,N,1)]*B.a[slice(j,N,N)]).sum()
10    return C;
11 }
```

問題	行けるかな?★	(UTPC 2008)
サイコロが巨大なすごろく (目の合計が 2^{31} まで) http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2107&lang=jp		

- U ターン禁止の表現: 有向辺 (道 + 向き) に番号を振り, ある番号 (が表す辺) から次にどの番号 (があらわす辺) に行けるかを表す, 遷移行列を作る

- n ターン後に行ける場所には, n 乗した遷移行列と初期位置を表すベクトルの積が対応する.
- 隣接行列などのグラフの表現は, 8 章に目を通した後に戻ってくることを推奨.

問題	Numbers★	(GCJ 2008 Round1A C)
概要: $(3 + \sqrt{5})^n$ の最後の 3 桁を求める http://code.google.com/codejam/contest/32016/dashboard#s=p2		

(参考書 [3, p. 239])

■GCJ への提出方法

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

問題	Leonard Numbers★★	(POI Training Camps 2008)
Fibonacci 数に似た, Leonard 数の和を求めよ http://main.edu.pl/en/archive/ontak/2008/leo		

第 13 章

配列操作

13.1 配列, `std::vector`, `std::array` と操作

データの列を表す方法と、操作を簡単に紹介する。

13.1.1 データ列の表現

```
C++
1 // 長さ 50 の整数の配列を用意し各要素を 0 で初期化 (A[0] から A[49] まで有効)
2 int A[50] = {};
```

C の配列は、定義の際に定数で長さを与える必要がある。この資料で扱う問題を解く場合は、必要な最大値を見積もって多少余分に取っておく。

C++ の場合は、`vector` を用いることで長さを実行時に決めることができる。

```
C++
1 #include <vector>
2 using namespace std;
3 int N = ...;
4 // 長さ N の整数の配列を用意し各要素を 0 で初期化 (A[0] から A[N-1] まで有効)
5 vector<int> A(N, 0);
```

```
Python3
1 n = 50
2 a = [0 for _ in range()]
```

C++ の新しい規格である C++11 では、`array` というデータ型が導入されている。これは `vector` と同様のインターフェースを持ち、一方定義の際に長さを定数で与える必要があるという点で、配列と `vector` の中間の性質を持つ。

```
C++11
1 #include <array>
2 using namespace std;
3 // 長さ 50 の整数の配列を用意し各要素を 0 で初期化 (A[0] から A[49] まで有効)
4 array<int, 50> A = {};
```

C++11 から導入された機能を使う場合は、このセミナーの iMac 環境では `g++` の代わりに `g++-mp-4.8` を用い、オプション `-std=c++11` を与える。すなわち `sample.cc` をコンパイルする場合は以下のように

なる。

```
$ g++-mp-4.8 -std=c++11 -Wall sample.cc
```

1

AJO に提出する場合も, C++ ではなく C++11 を選ぶ. C++11 で書く場合は生の配列よりも, `array` を使うほうがお勧めである. 関数の引数で受ける場合にも型が保存されることや, g++ 4.8 ではマクロ `_GLIBCXX_DEBUG` を定義しておくことで範囲外参照を発見できるなどのメリットがある.

C++11

```
1 #include <array>
2 int main() {
3     std::array<int, 3> a;
4     a[4] = 5;
5 }
```

`a[2]` までのところを `a[4]` にアクセス

```
$ g++-mp-4.8 -std=c++11 -Wall sample.cc
```

1

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

2

```
# 何も起こらない
```

3

```
$ g++-mp-4.8 -D_GLIBCXX_DEBUG -std=c++11 -Wall sample.cc
```

4

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

5

```
/.../c++/debug/array:152:error: attempt to subscript container
```

6

```
with out-of-bounds index 4, but container only holds 3
```

7

```
elements.
```

8

```
Objects involved in the operation:
```

9

```
sequence "this" @ 0x0x7fff6edfab00 {
```

10

```
type = NSt7__debug5arrayIiLm3EEE;
```

11

```
}
```

12

```
Abort trap: 6
```

13

メッセージは読みにくいですが異常検知に成功

13.1.2 列の操作

このような列に対して, 様々な操作が用意されている. どのような操作が用意されているかは言語による.

全ての要素を処理する

配列の全ての要素を処理する場合は, `for` 文を用いることが一般的である. 要素数 (この場合は 5) を間違えないように, 注意を払うこと.

C++

```
1 int A[5] = {0, 1, 2, 3, 4};
2 for (int i=0; i<5; ++i)
3     printf("
```

Python の `for` 文では, 添字を使う必要がない

Python3

```

1 A = [3,1,4,1,5]
2 for e in A:
3     print(e)

```

C++11 でも同様のことができる。

```

C++11 1 for (auto e:A) cout << e << endl;

```

整列 sort

3.1.1 節を参照。

反転 reverse

```

C++ 1 #include <algorithm>
    2 using namespace std;
    3 int A[5] = {3,5,1,2,4};
    4 int main() {
    5     sort(A,A+5);
    6     reverse(A,A+5); // 与えられた範囲を逆順に並び替え
    7     ... // cout に A を出力してみよう
    8 }

```

```

Python3 1 A = [3,1,4,1,5]
        2 b = list(reversed(A)) # A を反転させた b を得る (A は不変)
        3 A.reverse() # A そのものを変更

```

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

```

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

```

permutation の列挙

他に変わった機能として, C++ には `next_permutation` という関数がある. 典型的な使い方は以下のよう
に `do .. while` ループと組み合わせて, 全ての並び替えを昇順に列挙することである.

```
C++
1  #include <algorithm>
2      int A[4] = {1,1,2,3}; // あらかじめ昇順に並べておく
3      // sort(A, A+4); // 今回は昇順に並んでいるが初期値によっては必要
4      do {
5          cout << A[0] << A[1] << A[2] << A[3] << endl;
6      } while (next_permutation(A,A+4));
```

出力を確認せよ (実際に場合が尽くされているか?):

```
$ ./a.out
1123
1132
...
3121
3211
```

1
2
3
4
5
6

`next_permutation` は, まだ試していない組み合わせが (正確には全ての組み合わせを昇順に並べた時の次の要素が) ある場合, 配列の中身を並び替えて真を返す. そうでない場合は偽を返す. `next_permutation` が偽を返すとループが終了する. (配列の要素が昇順に並んでいない状態に初期化して (たとえば `int A[4] = {2,1,1,3};`), 上記のコードを実行すると, 出力はどのように変わるか?)

もしこの関数を自作する場合には, 間違える箇所が多いと予想されるので, 入念にテストを行うこと.

13.2 練習問題

問題

Hanafuda Shuffle

(国内予選 2004)

花札混ぜ切りのシミュレート: 初めに $1, \dots, N$ の数値を持つ N 枚のカードが一番下が 1, 一番上が N の順に整列されて山になっている. 図のような二つのパラメータ p, c で規定されるシャッフルを R 回行った後の, 一番上のカードを求めよ.

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1129&lang=jp>

■入力例と解釈 サンプル入力の

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

を解釈すると、まずカードは 5 枚、シャッフルは二回である。一度目で [5, 4, 3, 2, 1] が [3, 5, 4, 2, 1] となり二回目で [4, 3, 5, 2, 1] となる。従って、山の一番上は 4。

■入力とプログラムの骨組み まずは入力部分を作り、正しく読み込めていることを、 n や p, c を表示することで確認すると良い。続いて、山を整数の列 (配列または `vector`, `array` など) で表現する。シャッフルは回転 (`rotate`) で実装すると良い。

```
C++
1  int N, R, p, c;
2  int main() {
3      while (cin >> N >> R && N) {
4          // 山全体を作る
5          // 作った山を表示してみよう
6          for (int i=0; i<R; ++i) {
7              cin >> p >> c;
8              // シャッフル p, c を行う
9              // シャッフル毎に山全体を表示してみよう
10         }
11         // 山の先頭を出力
12     }
13 }
```

問題

Rummy

(UTPC2008)

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

カードは、赤緑青の三色で、数字は 1-9 まで。

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

■入力例と解釈

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

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

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

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

■解答方針 人間が役に当てはまるかを検討する場合、賢い (比較的複雑な) 試行を行うことで比較的少ない試行錯誤で最終的な判断にいたる、と想像される。「人間がどう考えるか」をコンピュータ上で実現すること

は人工知能の興味深い目標となることが多いが、コンピュータにとって最も簡単な方法は別に存在することも多い。(例: 122334 という並びを見て、慣れた人間は一目で 123, 234 と分解できるが、この操作を例外のない厳密なルールにできるだろうか?)

ここでは、(1) カードの全ての並び替えを列挙する (2) 各並び順につき、前から 3 枚ずつセットになっていることを確かめる、という方針で作成しよう。このような、「単純な試行で全ての可能性を試す」アプローチはコンピュータで問題解決を行う時に適していることが多い。

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

```
C++
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int T, card[16];
5  int main() {
6      cin >> T;
7      for (int t=0; t<T; ++t) {
8          for (int i=0; i<9; ++i) {
9              cin >> card[i];
10             card[i]//を出力
11         }
12         string color;
13         for (int i=0; i<9; ++i) {
14             cin >> color;
15             color//を出力
16         }
17     }
18 }
```

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

```
C++
1  cin >> T;
2  for (int t=0; t<T; ++t) {
3      for (int i=0; i<9; ++i) {
4          cin >> card[i];
5      }
6      string color;
7      for (int i=0; i<9; ++i) {
8          cin >> color;
9          if (color == "G") card[i] += 10;
10         else if (color == "B") card[i] += 20;
11     }
```

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


```

11         card[i] //を出力して確認
12     }
13 }

```

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

```

C++
1 bool is_good_set(int a, int b, int c) {
2     return is_same_number(a, b, c) || is_sequence(a, b, c);
3 }

```

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

```

C++
1 bool is_same_number(int a, int b, int c) {
2     // aとbとcが同じなら真, それ以外は偽
3 }

```

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

```

C++
1 bool is_sequence(int a, int b, int c) {
2     // a+2 と b+1 と c が等しければ真, それ以外は偽
3 }

```

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

テスト例:

```

C++
1 int main() {
2     // is_same_number のテスト
3     cout << is_same_number(3, 4, 5) << endl; // 偽
4     cout << is_same_number(3, 3, 3) << endl; // 真
5     // is_sequence のテスト
6     cout << is_sequence(3, 4, 5) << endl; // 真
7     cout << is_sequence(3, 3, 3) << endl; // 偽
8     // is_good_set のテスト
9     cout << is_good_set(3, 4, 5) << endl; // 真
10    cout << is_good_set(3, 3, 3) << endl; // 真
11    cout << is_good_set(3, 3, 30) << endl; // 偽
12    cout << is_good_set(5, 4, 3) << endl; // 偽
13 }

```

■勝利状態の判定 global 変数 card が勝利状態にあるかどうかを判定する

```

C++

```

```

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

```

■全体の組み立て 全ての並び順を作るコードと `is_all_good_set()` を組み合わせると、ほぼ完成である。

```

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

```

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

13.3 いろいろな問題

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&lang=jp		

注: 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間に合わないので、工夫した方法が必要である。

ヒント:

問題	かけざん	(夏合宿 2012)
決められた手順が何ステップ続くかを判定する。(初めに、有限ステップで終了する証明を行うか、無限に続く例を作成してみよう。) http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2424&lang=jp		

問題	Starting Line	(夏合宿 2011)
<p>エンジンで加速しながら、ゴールまで走ろう。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2298&lang=jp</p>		

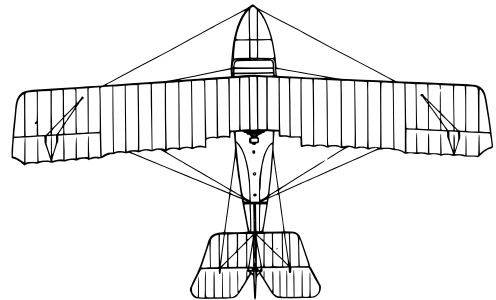
問題	Water Tank★	(国内予選 2004)
<p>いくつかの仕切りのある水槽がある。指定された位置と時刻の水位を求める。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1133&lang=jp</p>		

問題	Ants★	(Polish Collegiate Programming Contest 2011)
<p>2 匹のアリが木 (tree) の上を一定速度で歩いている。二回目に出会う時刻を求めよ。</p> <ul style="list-style-type: none"> ● 片方はもう片方の二倍の速さで進む ● 枝を下がる場合は上がるより二倍の速さで進む ● 根 (root) から、それぞれ別の方向にスタートする ● アリ同士がぶつかると向きを反対にして進む ● 根に到達した場合は裏に回ってから向きを反対にして進む ● 木の情報は、片方の蟻がたどる枝の順の上り/下りで与えられる。 (大きいのでメモリ上には保持できない) <p>https://szkopul.edu.pl/problemset/problem/hKLVBaShgwgjH3R0nMREDw8g/site/</p>		

main.edu.pl のアカウントは AOJ とは別に作成が必要である (初回のみ)。“Form/Year (a number)” はポーランド語バージョンの Klasa/Rok の英訳が Class/Year となっているので学年を指すかもしれない。

問題	Pilot★	(17th Polish Olympiad in Informatics)
<p>パイロットがどの程度まっすぐ飛べるかどうかを計算する。時刻毎の位置 (一次元) が数列 a_i として与えられる。数列の長さは最大 3,000,000 である。ずれてよい範囲として t が与えられる。数列の範囲 $[i, j]$ のうち、$a_k - a_l \leq t, \forall k, l, i \leq k, l \leq j$ という条件をみたす最大の長さを求めよ。</p> <p>https://szkopul.edu.pl/problemset/problem/lcU5m2RAICwNHsdzydb8JTQw/site/</p>		

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



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.

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

問題

Guesswork**

(11th Polish Olympiad in Informatics)

9 個の数が順に与えられるので, 一つずつ与えられた時点でそれが全体の何番目かを答える. 9 個すべてに回答した時点ですべて正解だったら勝ち. 勝率を (100 点がとれるくらいまで) 最大化する戦略を実現せよ.

<http://main.edu.pl/en/archive/oi/11/zga>

(この問題は現在 judge が動いていない)

難易度が高い問題は, 一学期履修した後に取り組むことが適切な場合もある.

第 14 章

整数と連立方程式

14.1 素因数分解、素数、ユークリッドの互除法など

例題	Prime Factorize	(AOJ)
整数を素因数分解する http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=NTL_1_A&lang=jp		

小さい数から順に割り切れるか試すと良い。被除数の平方根より大きな数で割る必要はない(なぜか?)

例題	Greatest Common Divisor	(AOJ)
最大公約数を求める。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_1_B&lang=jp		

ユークリッドの互除法というアルゴリズムが定番である。上記の問題の解説や参考書 ^{攻略}[2, pp. 441–443], 参考書 [3, pp. 107–] 参照。最大公約数を利用して簡単に最小公倍数も求められる (http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=NTL_1_C&lang=jp)。

例題	Sum of Prime Numbers	(PC 甲子園 2004)
与えられた数 n に対して、素数を小さい方から並べた時に n 番目の素数までの和を出力せよ。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0053&lang=jp		

エラトステネスの篩というアルゴリズムが定番である。はじめに、数が素数かどうかを記録する大きな配列

を用意し、すべての数が素数であるとしておく。2 の倍数を 2 を除いて消去する、3 以外の 3 の倍数を消去する、5 以外の 5 の倍数を消去する。というように「まだ消されていない中で最大の数を素数として残した後、その倍数を消去する」という手順を繰り返す。判定したい最大の素数の平方根より大きな素数については、その倍数を消す必要はない。参考書^{攻略}[2, pp. 438–439] 参照。

問題

Galaxy Wide Web Service

(夏合宿 2009)

Web サービスに、様々な惑星から周期的なアクセスがある。惑星 i の、一日の時間は $1 \leq d_i \leq 24$ 時間で、今の時刻は t_i 時である。各惑星からのアクセス量はその惑星の時刻のみに依存して決まる。(幸いなことにどの惑星の時刻も地球の 1 時間が単位である。) 最もアクセスが多くなる時間帯のアクセス量を求めよ。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2162&lang=jp>

方針:

- もし全周期求められればだいたい分かる

サンプル入力のように (1 2 3 4) と (2 1) という惑星であれば、

地球時間	0	1	2	3	4	5	6	7
惑星 1	1	2	3	4	1	2	3	4
惑星 2	2	1	2	1	2	1	2	1
合計	3	3	5	5	3	3	5	5

と、アクセス数は 4 時間の周期の繰り返しとなるので答えは 5

- 全周期は? 惑星の時間数の最小公倍数。1 から 24 まで全ての時間が揃うと、5354228880 になる (無理)
- 13, 17, 19, 23 を特別扱いする。残りの数の最小公倍数は 55440 となる。この時刻であれば列挙可能である。特別扱いした 4 つの数は、他のどの周期とも互いに素なので互いの最大値が重なる時刻がやがてあらわれる。つまり別に最大を求めて、全ての和を取ると良い。

入力と整形: 1-24 時間周期の惑星毎にアクセス数をまとめる (t 時間のずれも吸収する)

C++

```

1   while (cin >> N && N) {
2       int Q[25][25] = {{0}}; // Q[d][i] d 周期の i 時間目の量 (0<=i<d)
3       for (int i=0; i<N; ++i) {
4           cin >> d >> t;
5           for (int j=0; j<d; ++j) {
6               cin >> q;
7               Q[d][(j+d-t)%d] += q;
8           }
9       }
10      //この辺で答えを計算する
11  }
```

Ruby

```

1 while true
2   $N = gets.to_i
3   break if $N == 0
4   $Q = {}
5   (1..$N).each {
6     d, t, *q = gets.split("_").map{|s| s.to_i}
7     q.rotate!(t)
8     unless $Q[d]
9       $Q[d] = q
10    else
11      (0..d-1).each {|i| $Q[d][i] += q[i] }
12    end
13  }
14  # この辺で答えを求める
15 end

```

ここまでの処理で、 $1 \leq d \leq 24$ に対して $Q[d][0]$ から $Q[d][d-1]$ にその周期のアクセス数が格納されているので、適宜組み合わせる。

C++

```

1 const int L = 16*9*5*7*11;
2 int sum = 0, T[L] = {0};
3 for (int d=1; d<=24; ++d) {
4   if (d == 13 || d == 17 || d == 19 || d == 23 || d == 1)
5     sum += /*(Q[d][0]からQ[d][d-1]の最大値)*/;
6   else
7     for (int i=0; i<L; ++i) T[i] += Q[d][i]
8   }
9   cout << sum + /*(T[0]からT[L-1]の最大値)*/ << endl;

```

Ruby

```

1 L = 16*9*5*7*11
2 # 答えを計算するあたり
3 sum = 0
4 $T = Array.new(L, 0)
5 $Q.each{|d, q|
6   if d == 13 || d == 17 || d == 19 || d == 23 || d == 1
7     sum += q.max
8   else
9     (0..L-1).each {|i| $T[i] += q[i]
10    end
11  }
12  # sumと$T.maxが答え

```

この問題では 1 から 24 までという制約があったために最小公倍数をあらかじめ手で計算したが、計算機を用いて計算する場合はユークリッドの互除法(参考書 [3, pp. 107–]) を用いると良い。

C++

```

1 int gcd(int a, int b) {
2   if (a < b) swap(a, b);

```

```

3   return b == 0 ? a : gcd(b, a
4 }

```

問題

Extended Euclid Algorithm

(AOJ)

与えられた 2 つの整数 a, b について $ax + by = \gcd(a, b)$ の整数解 (x, y) を求めよ。

http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=NTL_1_E&lang=jp

拡張ユークリッド互除法で求められる。ユークリッド互除法が行っている計算で付随して求められる情報を活用する。



逆元

数え上げなどで「答えを $M = 10^9 + 7$ で割った余りを求めよ」という指示を持つ問題がある。そのような場合に、 $(p + q) \% M = ((p \% M) + (q \% M)) \% M$, $(p \cdot q) \% M = ((p \% M) \cdot (q \% M)) \% M$ などの関係を利用して、途中段階から剰余を取ると良い。多倍長整数などを用いることなく、効率的に計算できるためである。ここで除算については、 $(p/q) \% M = (p \cdot q^{-1}) \% M$ と変換する必要がある。 q^{-1} とは、 $(q \cdot q^{-1}) \% M$ を満たすような整数、 q の逆元である。整数 q と M が互いに素である時はつまり $\gcd(q, M) = 1$ であるので、 (q, M) についての拡張ユークリッド互除法により、 q^{-1} を得ることが出来る。

問題

中国剰余定理

(古典)

「3 で割ると 2 余り、5 で割ると 3 余り、7 で割ると 2 余る数はいくつか」一般に互いに素な整数 a, b, c に対する剰余 a', b', c' に対して、「 a で割ると a' 余り、 b で割ると b' 余り、 c で割ると c' 余る数」を一つ求める関数を作成せよ。

オンラインジャッジに適切な問題がなさそうなので、この問題は各自で入出力を作ってテストすること。テストしたデータをソースコードに添えること。

巨大な数を伴う計算を行う場合に、すべてを多倍長整数で計算すると計算コストが大きい。代わりに、複数の素数に対する剰余を計算して、最後に復元すると計算の効率が良い。囲碁 (19 路盤) の合法局面の数^{*1}は、このように求められた。

^{*1} 208168199381979984699478633344862770286522453884530548425639456820927419612738015378525648451698519643907259916015628128546089888314427129715319317557736620397247064840935 (DOI:10.1007/978-3-319-50935-8_17)

14.2 連立方程式を解く

14.2.1 言語機能: valarray

数値ベクトル (行列の行, あるいは列など) を扱う場合は `valarray` が便利である.

```
C++
1 #include <valarray>
2 valarray<int> V(0, N); // 要素数 N の整数のベクトルを作成, 各要素は 0
3 V = 3; // 全要素を 3 に
4 V[0] = 1;
5 V *= 10; // 全要素を 10 倍に
6 V
7
8 // 最大値 最小値 合計
9 cout << V.max() << ' ' << V.min() << ' ' << V.sum() << endl;
10
11 valarray<int> U(2, N);
12 V += U; // 各要素の和
13
14 // slice(initial, length, step)
15 V[slice(1,3,2)] = -3; // V[1] = V[3] = V[5] = -3
```

14.2.2 連立方程式

問題

Awkward Lights

(アジア地区予選 2010)

ライトがグリッド状に並んでいる. あるライトのスイッチを押すとそのライトに加えてマンハッタン距離 d にあるライトの on/off の状態が変わる. 全部のライトを消すことができるか?

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1308&lang=jp>

考え方: 2 回以上スイッチを押しても意味がないので, 各スイッチは押すか押さないかのどちらか. 各スイッチに 0 から $MN - 1$ の番号を振り, 押したかどうかを $x_i \in \{0, 1\}$ であらわす. 行列 A は MN 行/列で, 要素 a_{ij} は, スイッチ i を押すとライト j に影響があることを示すとする. 行列 A とベクトル x の積は, 変化を受けたスイッチを表すベクトルとなる. 各ランプが最初についていたかどうかを b_i で表すとして, $Ax = b$ に解があるかどうかを求める (ただしすべての演算を mod 2 で行う).

掃き出して行って行列左下の三角部分を 0 にした時に, どのスイッチでも消せないライトが残らなければ成功

入力

```
C++
1 typedef valarray<int> array_t;
2 int M, N, D;
```

```

3  int main() {
4      while (cin >> M >> N >> D && M) {
5          valarray<array_t> A(array_t(0, M*N+1), M*N);
6          for (int i=0; i<M*N; ++i) {
7              cin >> A[i][M*N];
8              for (int j=0; j<M*N; ++j) {
9                  int x0=i
10                 int d=abs(x0-x1)+abs(y0-y1);
11                 if (d==0 || d==D) A[i][j] = 1;
12             }
13         }
14         cout << solve(A) << endl;
15     }
16 }

```

A は MN 行, $MN+1$ 列の行列で, 右端の列は最初についているかどうか, 他の列の要素は i 個目のスイッチを押したときに j 個目のライトが切り替わるかを表す. $A[i]$ は i 行目を表し, 各行は `valarray<int>` 一つに対応する. (慣れたら行列全体をひとつの `valarray<int>` で表したほうが効率が良い)

計算

```

C++ 1  int solve(valarray<array_t>& A) {
2      int p = 0; // top → bottom
3      for (int i=0; i<M*N; ++i) { // left → right
4          int r = // 「p以降で A[r][i]!=0 の行」
5          if (/*適切な r が存在しなければ*/ continue;
6          swap(A[p], A[r]);
7          for (int j=p+1; j<M*N; ++j)
8              if (A[j][i]) {
9                  // 行 A[j] に行 A[p] を足しこむ, A[j]%=2 を忘れずに
10             }
11         ++p;
12     }
13     for (int r=0; r<M*N; ++r)
14         if (/* A[r][M*N] が非0で, A[r] の要素がそれ以外0だと*/ return 0;
15     return 1;
16 }

```

問題

Strange Couple★

(夏合宿 2009)

道路と交差点の一覧が与えられる. 標識がない交差点では U ターンも含めて等確率でランダムに道を選び, 標識がある交差点ではそこから目的地までの最短で到達できるルート (複数あればランダムに選ぶ) 人がいる. 出発地から目的地までに通る距離の合計の期待値を求めよ.

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2171&lang=jp>

最短路と連立方程式: まず, ダイクストラ法などで目的地から各交差点までの距離を求める. 続いて, 各交差点から出発した場合の期待値を変数として, 隣合う交差点から出発した場合の期待値との関係から連立方程式を立てる.

最短路の求め方は後日扱うので, まだ学んでいない場合は後で取り組むと良い.

14.3 その他の練習問題

問題	Afternoon Tea★	(Polish Olympiad in Informatics 2011)
紅茶とミルク、どちらをより多く飲んだか? http://main.edu.pl/en/archive/amppz/2011/her		

問題	Making Change	(codechef)
ある国の硬貨の価値は $D[1], \dots, D[N]$ である. 金額 C (とても大きい) をこの国の硬貨で支払う方法は何通り? ただし $D[1], \dots, D[N]$ は相異なりどの 2 つも互いに素. http://www.codechef.com/problems/CHANGE		

第 15 章

補間多項式と数値積分

15.1 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:

$$L(x) = \frac{(x-b)(x-c)}{(a-b)(a-c)}v_a + \frac{(x-a)(x-c)}{(b-a)(b-c)}v_b + \frac{(x-a)(x-b)}{(c-a)(c-b)}v_c. \quad (15.1)$$

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 15.2 節へ進んで良い。)

問題

Find the Outlier

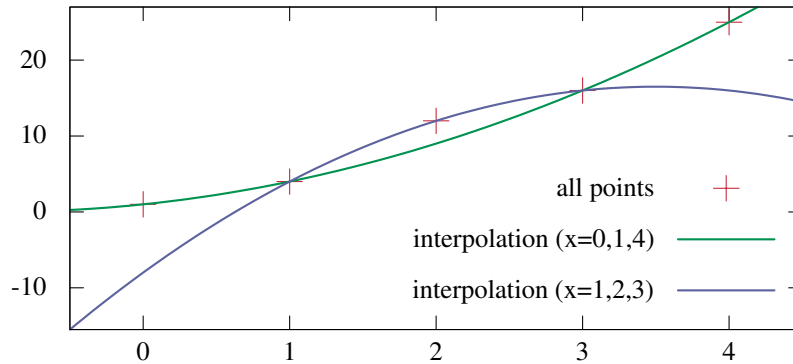
(アジア地区予選 2012)

秘密の多項式 $f(x)$ の次数 d と $d+3$ 個の点, $(0, v_0), (1, v_1), \dots, (d+2, v_{d+2})$ 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&lang=jp>

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 $\text{interpolate}(n, E)$ は以下のよう

C++

```

1  int D;
2  double V[16];
3  double interpolate(int n, int E) {
4      // L(n) を補間多項式により計算する
5      // ただし点 (n, V[n]) と (E, V[E]) は使わない
6      double sum = 0.0;
7      for (int k=0; k<D+3; ++k) {
8          if (/*kがnかEなら*/ continue;
9          double p = V[k];
10         for (int i=0; i<D+3; ++i)
11             if (! (/*iがkでもnでもEでもなければ*/)
12                 p *= (...-i) / (double) (k-i);
13         sum += p;
14     }
15     return sum;
16 }
```

Python3

```

1  # 配列 V に値が入っているとする
2  def interpolate(n, e):
3      total = 0.0
```

```

4   for k in range(len(V)):
5       if ... # k が n か E なら
6           continue
7       p = V[k]
8       for i in range(len(V)):
9           if ... # i が k でも n でも E でもなければ
10              p *= 1.0*(n-i)/(k-i)
11       total += p
12   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++

```

1 bool outlier(int E) {
2     for (int i=0; i<D+3; ++i) {
3         if (i==E) continue;
4         double p = interpolate(i, E);
5         if (/*pとV[i]の絶対値(abs)が0.1より離れていたら*/)
6             return false;
7     }
8     return true;
9 }

```

Python3

```

1 def outlier(e):
2     for i in range(len(V)):
3         if i == e:
4             continue
5         p = interpolate(i, e)
6         if ... # p と V[i] の絶対値(abs) が 0.1 より離れていたら
7             return False
8     return True

```

C++

```

1 #include<iostream>
2 #include<cmath>
3 // この辺に上記の関数を定義する
4 int main() {
5     while (cin >> D && D) {
6         for (int i=0; i<D+3; ++i) cin >> V[i];
7         for (int i=0; i<D+3; ++i)
8             if (outlier(i)) { // i を無視すれば全て整合する
9                 cout << i << endl;
10                break;
11            }
12    }
13 }

```

Python3

```

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

```

15.2 数値積分とシンプソン公式

例題

Integral

(PC 甲子園 2003)

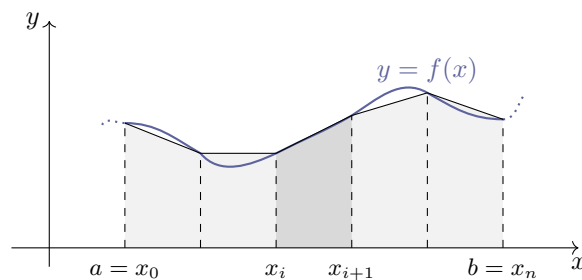
長方形の和を用いて面積の近似値を求めよ.

<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)

15.2.1 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?

15.2.2 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 (15.1)) 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 軸で区間に分けて、各区分事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。各区分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*

(アジア地区予選 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&lang=jp>

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から、その面積を x 軸方向に積分して体積を求める。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 z 軸の幅を求める関数 $\text{width}(\text{polygon}, x)$ は、たとえば以下のように入力された多角形の辺を一巡して求めることができる。


```

Python3 1 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
2     w = []
3     for i in range(len(polygon)):
4         p, q = polygon[i], polygon[(i+1)]
5         if p[0] == x:
6             w.append(p[1])
7         elif (p[0] < x and x < q[0]) or (p[0] > x and x > q[0]):
8             x0, y0 = p[0], p[1]
9             x1, y1 = q[0], q[1]
10            w.append(y0 + 1.0*(y1-y0)*(x-x0)/(x1-x0))
11    assert len(w) > 0
12    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 `P1` は多角形 `P1` の (x,y) 座標を順に格納した配列、`P2` 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 座標を昇順に格納した配列とする。

```

Python3 1 def volume(Pxy, Pxz, X):
2     X = sorted(set(X))
3     total = 0.0
4     xmin = max(min(Pxy)[0], min(Pxz)[0])
5     xmax = min(max(Pxy)[0], max(Pxz)[0])
6     for i in range(len(X)-1):
7         a, b = X[i], X[i+1]
8         if not (xmin <= a and a <= xmax and xmin <= b and b <= xmax):
9             continue
10        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

```

15.3 道具としての Fast Fourier Transform (FFT)

(この節の説明はまだ書かれていない)

問題	Best Position★	(Kuala Lumpur 2014)
<p>農耕と牧畜の生産が最大になる場所を探す。畑候補は長方形領域で、それ以内の畑パターンを重ねあわせた時に一致したマスで生産が生じる。</p> <p>https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=645&page=show_problem&problem=4820</p>		

畑候補と畑パターンをそれぞれ多項式として表現して、その積の多項式のある係数が、ある場所で重ねあわせた時の領域の和になるように工夫する。

問題	Point Distance★	(春コンテスト 2013)
<p>各セル (x,y) に、分子がいくつ入っているか C_{xy} で表される。すべての分子のペアを考えた時の平均距離と距離毎のヒストグラムを示せ。</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2560&lang=jp</p>		

二次元の FFT の練習。ヒストグラムは long long が必要。

問題	The Teacher's Side of Math★	(アジア地区予選 2007)
<p>解の一つが $a^{1/m} + b^{1/n}$ であるような、整数係数多項式を求めるプログラムを作成せよ。(a, b は互いに異なる素数、m, n は 2 以上の整数)</p> <p>http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1284&lang=jp</p>		

ヒント: 一般に 1 の原始 m 乗根を ξ , 1 の原始 n 乗根を η として

$$\prod_{j=0}^{m-1} \prod_{k=0}^{n-1} (x - a^{1/m} \xi^j - b^{1/n} \eta^k)$$

という多項式を作ると

$$= \prod_{j=0}^{m-1} \{(x - a^{1/m} \xi^j)^n - b\} = \prod_{k=0}^{n-1} \{(x - b^{1/n} \eta^k)^m - a\}$$

と変形できることを利用する。

数が小さいので、適当に展開しても良いし、FFT を用いても良い。

第 16 章

区間の和/最大値/最小値と更新

概要

ある程度大きなデータを与えられたうえで、質問ごとにデータの一部を調査するタイプの問題を考える。事前に準備をしておく(≈ 適切なデータ構造を用いる)ことで質問に効率的に答えられるようになる。ここでは一列あるいはグリッド上に並んだデータについて考える。

16.1 累積和

初めに、列 a_i の先頭からの和 $S_{i+1} = \sum_{k=0}^i a_k$ を活用する問題を考える。 $S_0 = 0$ とする。

例題

Maximum Sum Sequece

(PC 甲子園 2003)

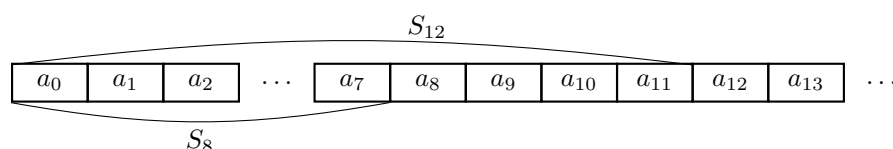
整数の並び $a_1, a_2, a_3, \dots, a_n$ で、連続した 1 つ以上の項の和の最大値を求めよ。

注: もし a_i が非負なら当然すべての和が最大なので、負の数を含む。負の数を含む区間が最大になることもある。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0022&lang=jp>

(これは maximum subarray problem という問題で、列の長さを N として $O(N)$ の解き方がある。しかし、ストーリーの都合で全区間を調べて $O(N^2)$ で解くことにする)

区間 $[i, j]$ の和、すなわち $\sum_{k=i}^{j-1} a_k$ を定義通りに計算すると、 $j - i - 1$ 回の加算が必要である。これを $j - i$ によらず 1 回の演算で済ませたい。それには、事前に $S_0 = 0, S_{i+1} = \sum_{k=0}^i a_k = S_i + a_i$ となるような補助変数 S を用意しておけば良い。これを用いると、区間 $[i, j]$ の和は $S_j - S_i$ で求められる。たとえば、 $a_8 + a_9 + a_{10} + a_{11} = S_{12} - S_8$ である。



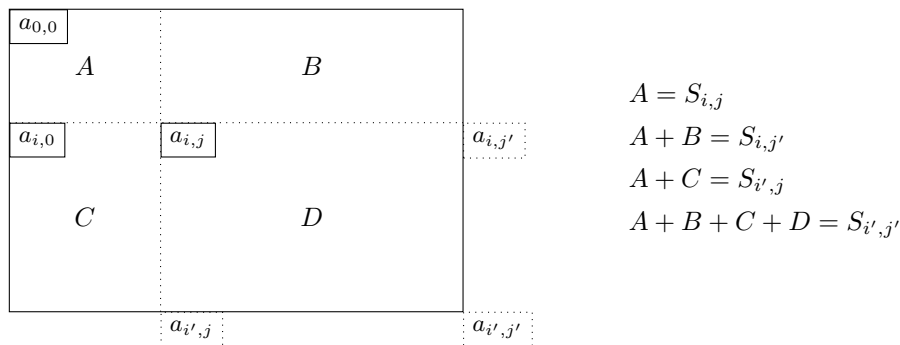
これを用いて、すべての区間の和を調べると求める最大値を得られる。

問題	A Traveler	(JOI 2009)
左右に行ったり来たりする旅程が与えられるので歩いた合計 (の剰余) を求めよ。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0549&lang=jp		

考え方: 旅程の要素に「左に 30 個目の宿場に移動する」という指示があったとして、30 回の加算をすることは避けたい。そこで事前に各宿場について左端からの距離を事前に計算しておく。すると旅程の要素数である M 回の加算で合計の距離を求められる。

二次元の累積和

同様の考え方は二次元にも応用できる: 補助変数を $S_{i+1,j+1} = \sum_{k=0}^i \sum_{l=0}^j a_{k,l}$ と定義する。縦 $[i,i')$, 横 $[j,j')$ の長方形領域の和は $S_{i',j'} - S_{i,j'} - S_{i',j} + S_{i,j}$ となる。(大きな長方形から余分にを引いて、引きすぎたものを調整する) つまり、長方形領域の面積によらない定数回の演算で求めることができる。



例題	Maximum Sum Sequence II	(PC 甲子園 2003)
和が最大になる長方形領域を知りたい。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0098&lang=jp		

(この問題も縦横どちらかの maximum subarray problem に変換することで、 $O(N^3)$ で解ける。しかし、ストーリーの都合で全長方形区間を調べて $O(N^4)$ で解く意図で掲載する)

問題	Planetary Exploration	(JOI 2010)
指定の長方形区域の資源の数を知りたい。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0560&lang=jp		

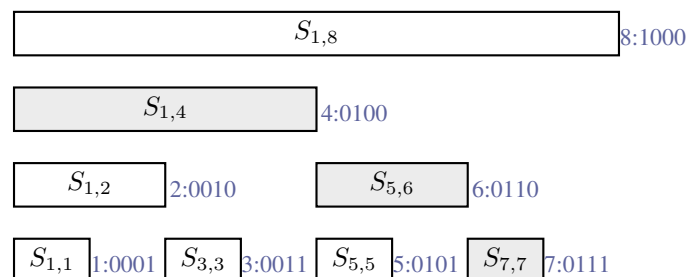
問題	Coffee Central★	(World Finals 2011)
カフェを開くののに最適な場所を知りたい。 https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=45&page=show_problem&problem=3133		

菱形の領域は 45 度回転させると扱いやすくなる場合がある。

16.2 Binary Indexed Tree (Fenwick Tree)

今度は処理の途中でデータ a_i の一部が変化する状況を考える。前節のように補助変数 S を使うと、質問には $O(1)$ で回答できて効率が良いが更新には $O(N)$ かかってしまう。質問回答に $O(\log N)$ のコストを許容することで、 $O(\log N)$ の更新を実現する手法を紹介する。(参考書 [3, pp. 160–])

問題	Range Query - Range Sum Query	(AOJ)
区間の合計値を管理せよ http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DSL_2_B&lang=jp		



まず図のような、2 のべき乗の長さの区間に対する和を求めておく。ここで $S_{i,j}$ という表記は区間 $[i,j]$ の和とする。前節までと表記が異なり、 j を含む。また、管理対象の列は a_1 と 1 から始まるとする。(0 から始めることもできるが、後を書くように微調整が必要である)

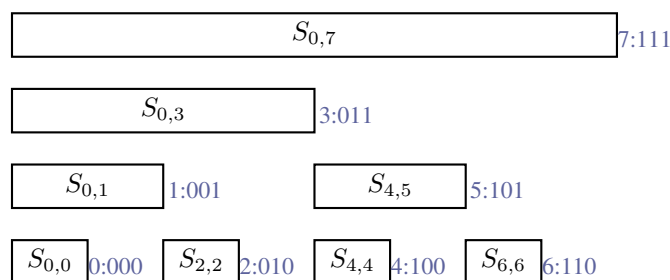
このような準備を行うと $x \geq 1$ にたいして $S_{1,x}$ を $\log x$ 個までの和として表現することができる。たとえば、 $S_{1,7} = S_{1,4} + S_{5,6} + S_{7,7}$ である。また、ある要素の値が変化した時には、最大 $\log N$ 個の部分 and を更新

すれば良い。たとえば、 a_7 が更新された時は、それを含む区間である $S_{7,7}$ と $S_{1,7}$ を更新する。

このようなデータを配列で管理するデータ構造が Fenwick tree または Binary indexed tree (BIT) である。青字で記した要素番号 (コロンの右側は 2 進表記) を用いると次のように管理することができる。 n 番目の配列の要素 $\text{BIT}[n]$ には、左端 l が n の最下位ビットを落とした数 +1 でまた右端 r が n であるような区間の和 $S_{l,r}$ を格納する。したがって、1 から n までの和は、まず $\text{BIT}[n]$ を加え、カバーする区間の左端 l が 1 に到達するまで、現区間の $l-1$ を右端とするような新しい区間を探して足すことを繰り返せば良い。そのような区間は n の最下位ビットを落とすことで求められる。また a_n を更新する場合は $\text{BIT}[n]$ の更新後に、その区間を含むブロックを (左端を減らさないように) 右端を広げることで探すことを繰り返す。

```
C++
1  int BIT[1000010], bit_size;
2  void bit_init(int n) {
3      fill(BIT, BIT+n, 0);
4      bit_size = n;
5  }
6  int bit_sum(int n) { // [1, n] の和
7      int ans = 0;
8      while (n > 0) { // 全体区間左端では 1 を一つだけ含むので最後は 0 になって終了
9          ans += BIT[n];
10         n &= n-1; // 最下位 (右端) の 1 を落とす: 左に移動
11     }
12     return ans;
13 }
14 void bit_add(int n, int v) { // n 番目の要素に v に加える
15     while (n <= bit_size) {
16         BIT[n] += v;
17         n += n & (-n); // 最下位 (右端) の 1 を繰り上げる
18     }
19 }
```

■参考: 0-index の場合



和: $n = (n \& (n + 1)) - 1$ ($n \geq 0$) 右に 0 のある 1 を繰り下げる

更新: $n |= n + 1$ ($n < \text{size}$) 一番右の 0 を 1 に

問題

引越し

(夏合宿 2012)

小さい順に並べ替えるのに必要な体力の合計を求める。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2431&lang=jp>

全部並べ替えると 1 から N までの総和なので、それから並べ替えなくて良い (=昇順の) 部分列の重さの総和の最大値を引けば良い。

i 番目の数字 x_i で終わるような昇順の部分列の重さの和の最大値を A_i と表記すると、

$$A_i = x_i + \max_{j < i, x_j < x_i} A_j$$

なので、数列を左から見てゆくと順に求められる。ここで i より小さい j について最大値を逐一調べると時間がかかるので、 x_i をキーとした BIT を使って効率良く処理を行う。

C++

```
1 long long N, x;
2 int main() {
3     cin >> N;
4     bit_init(N+1);
5     for (int i=0; i<N; ++i) {
6         cin >> x;
7         long long cost = // xまでの最大値
8         ... // xまでの最大値を cost+xに更新
9     }
10    cout << (1+N)*N/2 - bit_max(N) << endl;
11 }
```

Ruby

```
1 N = gets.to_i
2 X = gets.split('_').map{|s| s.to_i}
3
4 tree = BIT_Max.new(N+1)
5 X.each {|x|
6     cost = ... // xまでの最大値
7     ... // xまでの最大値を cost+xに更新
8 }
9 puts (1+N)*N/2 - tree.max(N)
```

区間の和を管理する BIT の代わりに、先頭からの区間 $[1, n]$ の最大値を持つデータ構造を考える。

C++

```
1 long long BIT[100010], bit_size; // long long は 64bit 整数.
2 void bit_init(int n) {
3     fill(BIT, BIT+n, 0);
4     bit_size = n;
5 }
6 long long bit_max(int n) { // [1, n] の最大値
```

```

7      long long ans = 0;
8      while (n > 0) {
9          ans = max(ans, BIT[n]);
10         n &= n-1;
11     }
12     return ans;
13 }
14 void bit_setmax(int n, long long v) { // # [1,n] の最大値を v に更新する
15     while (n < bit_size) {
16         BIT[n] = max(BIT[n], v);
17         n += n & (-n);
18     }
19 }

```

Ruby

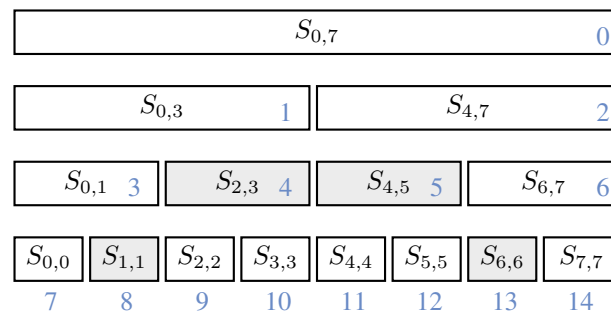
```

1 class BIT_Max
2     # 1-index
3     def initialize(n)
4         @array = Array.new(n+1, 0)
5     end
6     def max(n) # [1,n] の最大値
7         ans = 0
8         while n > 0
9             ans = @array[n] if ans < @array[n]
10            n &= n-1
11        end
12        ans
13    end
14    def setmax(n, v) # [1,n] の最大値を v に更新する
15        while n < @array.size
16            @array[n] = v if @array[n] < v
17            n += n & (-n)
18        end
19    end
20 end

```

16.3 Segment Tree と Range Minimum Query

次に区間の最小値や最大値について考えてみる。区間 $[i,j]$ の最小値は、残念ながら、区間 $[0,i]$ の最小値や区間 $[0,j]$ の最小値からは求めることができない。そこでもう少しデータの豊富な Segment Tree を用いる。(参考書 [3, pp. 154–]).



図のような全区間を根として番号 0 とする完全二分木で、各区間のデータを保持する。葉以外の各区間は 2 つの子を持ち、左の子が親の区間の左半分、右の子が残りを担当する。たとえば、区間 $[2, 6]$ の最小値は図の灰色部分のように、 $\min(S_{1,2}, S_{2,3}, S_{4,5}, S_{6,6})$ で求められる。どのような区間をとっても、調べる箇所は各深さ毎に最大 2 までである。

この実装では、ある id について、左の子は $id*2+1$ 、右の子は $id*2+2$ 、親は $(id-1)/2$ である。元の配列 a_i の長さを N とすると、 N が 2 のべき乗なら $2N$ の領域を、そうでなければ最大 $4N$ の領域を使用する。

また、根付き木の最小共通祖先 LCA (least/lowest common ancestor) を、深さ優先探索の訪問時刻に対する RMQ として、計算することもできる。参考書 [3, pp. 292–]

問題	Range Query - Range Minimum Query	(AOJ)
区間の最小値を管理せよ http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DSL_2_A&lang=jp		

問題	Balanced Lineup	(USACO 2007 January Silver)
牛が並んでいる。区間 $[a, b]$ で身長が最大の牛と最小の牛の差は? http://poj.org/problem?id=3264		

scanf 必要。

問題	Drawing Lots*	(UTPC2009)
あみだくじについて、選ばれた場所がアタリとなるように横線の数を追加するとして、その最小本数を求めよ。 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2192&lang=jp		

問題	Distance Queries★	(USACO 2004 February)
農場間 2 点の道のりを求める http://poj.org/problem?id=1986		

問題	Apple Tree★	(POJ Monthly–2007.08.05)
りんごの数を数える http://poj.org/problem?id=3321		

第Ⅲ部

補遺

付録 A

バグとデバッグ

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

A.1 バグの予防とプログラミング作法

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。

■変数の活用 変数を適切に使うとプログラムが見やすくなる

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。(デバッグや printf で表示しやすい)

■関数の活用 同様に見通しの良いプログラムとなる。

```
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; }

```

■定数の活用 定数にも名前をつけると見通しが良い。

```

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) // ああつ!
    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 }

```

■有名な落とし穴を避ける あらかじめ学んでおくの良いことがある。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 コンパイラは、時として親切な指摘をしていることがある。それらを見逃さないようにする。

- “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  }
```

A.2 デバッグの道具

A.2.1 道具: assert

プログラムの理解と実行時のテストのために C, C++, Python, Java などの主要言語で、assert という仕組みが用意されている。assert 文は、プログラム実行時に条件式を検査し、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。

■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。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、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」しておくとお見通しが良い。

このようにして factorial(-1) 等呼び出すと、エラーを表示して止まる。

```
Assertion failed: (n > 0), function factorial, file factorial.cc, line 3. 1
```

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

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

assert は Python でも使用可能である。python3 -O のようにオプション-O を付けて起動すると、assert 文は無効化される。

```
Python3
1 def factorial(n):
2     assert n >= 0
3     if n == 0:
4         return 1
5     return n*factorial(n-1)
```

assert は Java でも使用可能である。

```
Java
1 public class Main {
2     static int factorial(int n) {
3         assert n >= 0;
4         if (n == 0) return 1;
5         return n*factorial(n-1);
6     }
7     public static void main(String[] args) {
8         System.out.println(factorial(3)); // これは成功する
9         System.out.println(factorial(-3)); // これは...
10    }
11 }
```

Java の場合は、実行時オプションで-ea をつけると (つけたときのみ)assert が有効になる。

```
$ java -ea Main
6
Exception in thread "main" java.lang.AssertionError
    at Main.factorial(Main.java:3)
    at Main.main(Main.java:10)
```

A.2.2 道具: -fsanitize

比較的新しい C++ では、`-fsanitize=undefined`、`-fsanitize=address` などのコンパイルオプションが利用可能である。これらのオプションは、「不正なプログラムを書いたが、そのことが分からない、あるいは場所が特定できない」という時に役に立つ。

次のプログラムの関数 `fib` は引数 `i==0` の時に値を返さないバグがある。

```
C++
1 #include <iostream>
2 using namespace std;
3 int fib(int i) {
4     if (i == 1) return 1; // i==0 の場合を書き忘れ
5     else if (i >= 2) return fib(i-1)+fib(i-2);
6 }
7 int main() {
8     cout << fib(2) << endl;
9 }
```

以下のように `-fsanitize=undefined` オプションを有効にした場合に、実行時エラーが検知される。

```
$ g++ -Wall fib.cc
$ ./a.out
2
$ g++ -Wall -fsanitize=undefined fib.cc
$ ./a.out
fib.cc:3:5: runtime error: execution reached the end of a value-returning function without r
Abort trap: 6
```

次の(うっかりな)プログラムは、キーボードから読んだ整数が 0 以上 3 以下の場合にのみ正常動作する。

```
C++
1 #include <iostream>
2 using namespace std;
3 int A[4] = {0,1,2,3};
4 int f(int idx) {
5     return A[idx];
6 }
7 int main() {
8     int idx;
9     cin >> idx;
10    cout << f(idx) << endl;
11 }
```

`-fsanitize=undefined` をつけてコンパイルすると、5 行目で違反が起こったことを知らせてくれる。


```

$ g++ -Wall -fsanitize=undefined test.cc 1
$ ./a.out 2
3 3
3 4
$ ./a.out 5
8 6
test.cc:5:10: runtime error: index 8 out of bounds for type 'int [4]' 7

```

`-fsanitize=undefined` は便利だが万能ではない。メモリ関係の実行時エラーは `-fsanitize=address` で発見出来ることもある。以下のプログラムは `i=4` の時に不正なアドレスにアクセスする。

```

C++ 1 #include <iostream>
2 using namespace std;
3 int A[4]={0,1,2,3};
4 int main() {
5     int *p = A;
6     for (int i=0; i<5; ++i) // 5は4の誤り
7         cout << *(p+i) << endl;
8 }

```

`-fsanitize=undefined` をつけてもこの不正なメモリアクセスは検知されず、何事もなかったかのように実行される。

```

$ g++ -Wall -fsanitize=undefined addr.cc 1
$ ./a.out 2
0 3
1 4
2 5
3 6
8842826 7

```

`-fsanitize=address` をつけてコンパイルすると、不正なメモリアクセスが検知される。

```

$ g++ -Wall -fsanitize=address addr.cc 1
$ ./a.out 2
0 3
1 4
2 5
3 6
===== 7
==37898==ERROR: AddressSanitizer: global-buffer-overflow on address 0x000108f8d110 at pc 0x0000000000000000
READ of size 4 at 0x000107f8d110 thread T0 9
    #0 0x107f8c8e9 in main (a.out:x86_64+0x1000018e9) 10
    #1 0x7fff50332014 in start (libdyld.dylib:x86_64+0x1014) 11

```

A.2.3 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
```

A.2.4 道具: gdb

以下のように手が滑って止まらない `for` 文を書いてしまったとする.

```
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 7
(gdb) up // 何回か up して main に戻る 8
(gdb) up 9
#12 0x080486ed in main () at for.cc:6 10
6         cout << "hello_" << endl; 11
(gdb) list 12
1     #include <iostream> 13
2     using namespace std; 14
3     int main() { 15
4         for (int i=0; i<10; ++i) { 16
5             for (int j=0; j<2; ++i) 17
6                 cout << "hello_" << endl; 18
7                 cout << "world" << endl; 19
8             } 20
9         } 21
(gdb) p i 22
$1 = 18047 23
(gdb) p j 24
$2 = 0 25

```

主なコマンド:

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

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

A.2.5 道具: valgrind

```

C++ 1 int main() {
    2     int p; // 初期化忘れ
    3     printf("
    4 }

```

gdb を用いる時と同様に `-g` オプションをつけてコンパイルする。

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

実行時は、valgrind コマンドに実行プログラムを与える。

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

```

A.3 標本採集: 不具合の原因を突き止めたら

バグの原因を特定したら、標本化しておくで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で役に立つだろう。

配列の境界

```
C++ 1    int array[3];
    2    printf("

```

初期化していない変数

```
C++ 1    int array[3];
    2    int main() {
    3        int a;
    4        printf("
    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 溢れ

```
C++ 1    int main() {
    2        int a[1000000000]; // global 変数に移した方がよい
    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("

```

// 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("

```

```

C++ 1  int a, b;
    2  cin >> a, b;

```

付録 B

プログラミング言語と環境の理解

B.1 ループ不変条件

以下のような、簡単な `for` 文の正しさについて考えてみよう。

<code>def</code> 名前 (仮引数1, 仮引数2...):	1
変数 = 初期値	2
for ...	3
変数 = 式 # 変数を新しい値に書き換え	4
return 変数	5

以下の例題と回答例を考える。

例題

連続する数の和 (`sum_to_n`)

(judge なし)

整数 1 から n までの合計を、愚直に足して求める関数 `sum_to_n(n)` を作成せよ。ただし $1 < n$ とする。(n=10000 など検算する)

回答例:

Python3	1	<code>def sum_to_n(n):</code>
	2	<code>sum = 0</code>
	3	<code>for i in range(1, n+1):</code>
	4	<code>sum = sum + i</code>
	5	<code>return sum</code>

このようなプログラムがどのように導かれるか追ってみよう。

仮に固定値である 3 までの合計であれば、シンプルに記述可能である。

Python3	1	<code>def sum3():</code>
	2	<code>return 1+2+3</code>

補助変数 s_i (0 から i までの和) を用いて、等価なプログラムに書き直すこともできる。

```

Python3 1 def sum3():
2     s0 = 0 # 空集合の和
3     s1 = s0 + 1 # 1 の和
4     s2 = s1 + 2 # 1 と 2 の和
5     s3 = s2 + 3 # 1 と 2 の 3 の和
6     return s3

```

代入を用いると、補助変数 s_i を一つの変数 `sum` で表現することもできる。

```

Python3 1 def sum3():
2     sum = 0 # sum は s0 相当
3     sum = sum + 1 # 左辺の sum は s1, 右辺の sum は s0 相当
4     sum = sum + 2
5     sum = sum + 3
6     return sum # sum は s3 相当

```

これを `for` で書き換えたものが、冒頭の回答例である。

```

Python3 1 def sum_to_n(n):
2     sum = 0
3     for i in range(1, n+1):
4         # (a) この時点で sum の値は 0 から i-1 までの和
5         sum = sum + i # なお, sum += i と書いても同じ
6         # (b) この時点で sum の値は 0 から i までの和
7     return sum

```

コメント (a)(b) に記述した `sum` の値に関する条件が、ループのどの時点でも成り立つことを確認してほしい。このような条件を、ループ不変条件という。これらの `sum` の値に関する条件を用いて、関数が n までの和を計算することを証明することができる。

B.2 再帰

B.2.1 帰納的定義と線形再帰

「何かを繰り返して計算したい」状況を考える。while や `for` のような反復・ループによって書く方法と、関数の中から自分自身を呼び出す再帰 (recursion) によって計算する方法がある。ここでは後者について掘り下げる。

例として、 $1 + 2 + \dots + n$ に相当する関数を、漸化式で定義してみよう。この関数を $\text{sum}(n)$ と書くことにすると、次のように帰納的に定義できる。

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

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 Python や 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。

Python3

```

1 def sum1(n):
2     if n == 1:
3         return 1
4     else:
5         return n + sum1(n-1)

```

■プログラムの正しさの理解 次の二点を確認すると良い

- basecase の正しさを確認: $n = 1$ のとき $\text{sum1}(n) = 1$
- $n - 1$ まで正しいと仮定して n の正しさを確認: $\text{sum1}(n) = n + \text{sum1}(n-1)$

B.2.2 枝分かれを伴う再帰

ここまで見た再帰呼び出しでは、1 つの関数からその関数を 1 回だけ呼び出していた。これ以降は、複数の再帰を呼び出す場合を扱う。

例題

フィボナッチ数

(judge 無し)

フィボナッチ数列 (Fibonacci numbers) とは、

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

のように、「前 2 つの数の和」で作られる数の列である。 n 番目のフィボナッチ数 $\text{fib}(n)$ を帰納的に定義せよ。また、関数 $\text{fib}(n)$ を定義せよ。

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)

```

なお再帰は fibonacci 数を計算することに関しては最善の方法ではない。より効率良くフィボナッチ数を求める方法は別に扱う。

■末尾再帰 (tail recursion) 式を少し変形した、次の定義の $\text{sum}'(s, n)$ を考える。

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

上記の定義中の s は、「これまでの和」に相当すると考えて、 $\text{sum}(n) = \text{sum}'(0, n)$ であることを確認せよ。

```
Python3 1 def sum2(s, n):
2     if n == 1:
3         return s+1
4     else:
5         return sum2(s+n, n-1)
```

返り値として自分自身を呼ぶ構造を末尾再帰と呼ぶ。このように書いておくと、コンパイラが最適化しやすいというメリットがある。

また、 s のような補助的な変数を導入すると再帰の見通しがよくなることがある。



maximum recursion depth exceeded

再帰関数は、どこかで止まるように実装する必要がある。たとえば以下の関数は止まるところが定義されていない。

```
Python3 1 def inf(a):
2     return 1+inf(a)
```

この関数を $\text{inf}(0)$ と呼び出すと、

```
>>> inf(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in inf
  File "<stdin>", line 2, in inf
  File "<stdin>", line 2, in inf
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

というように、`RecursionError: maximum recursion depth exceeded` というエラーで処理を続行できなくなった旨表示される。

■動作に関する再帰的な考え方 例題として 1 から n までの整数を表示するという動作 $\text{print_range}(n)$ を考える。先程の sum と同様に以下のように動作を定義できる。

$$\text{print_range}(n) = \begin{cases} 1 \text{ を表示} & n = 1 \\ \text{print_range}(n - 1) \text{ (後に) } n \text{ を表示} & \text{それ以外} \end{cases}$$

```
Python3 1 def print_range(n):
2     if n == 1:
3         print(1)
4     else:
```

```

5     print_range(n-1)
6     print(n)

```

例題

m 進数

(judge 無し)

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関数 $\text{mdigit}(m, n)$ を作成せよ.

```

>>> mdigit(3,2)
00
01
02
10
11
12
20
21
22

```

1
2
3
4
5
6
7
8
9
10

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. つまり, $n - 1$ 桁の 2 進数を全部表示する関数を誰かが作ってくれていたとして (*1), それを元に n 桁の m 進数を全部表示する関数を作れるだろうか.

n 桁の 2 進数は左端の 1 文字 (0 または 1) と残り $n - 1$ 桁の 2 進数に分解できる. そこで, 左端に 0 を追加して $n - 1$ 桁の 2 進数を全部表示する, 左端に 1 を追加して $n - 1$ 桁の 2 進数を全部表示する, というようなことをしたい.

上記を踏まえて (*1) の関数を微調整して, 左端に書いてほしい文字列 prefix を引数に追加し, 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 $B(\text{prefix}, n)$ とする.

$$B(\text{prefix}, n) = \begin{cases} \text{prefix を表示} & (n = 0) \\ B(\text{prefix} + 0, n - 1) \text{ と } B(\text{prefix} + 1, n - 1) \text{ を実行} & (n > 1) \end{cases} \quad (\text{B.1})$$

prefix	3	2,1	
10	0	00	← B(100,2) を通じて実現
10	0	01	
10	0	10	
10	0	11	
10	1	00	← B(101,2) を通じて実現
10	1	01	
10	1	10	
10	1	11	

$B(10, 3)$ の動作: $B(100, 2)$ と $B(101, 2)$

$B(0, n)$ が, n 桁の 2 進数を全て表示することを確認せよ. 2 進数だけでなく m 進数に対応するには, 0 と 1 のみ分岐させていた部分を, for 文などで 0 から $m-1$ まで分岐させれば良い.

prefix は文字列 (str) を想定して説明したが, int で表現しても良い. その場合, たとえば, prefix の右に 1 を加える操作は, $\text{prefix} * 10 + 1$ となる.

文法: Python 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.

```
Python3 1 >>> print("{:05d}".format(3))
2 00003
```

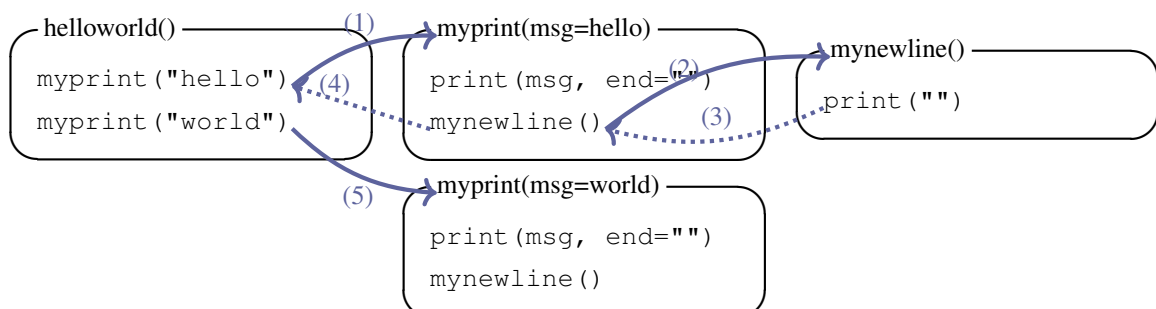
B.2.3 関数と実行状態の管理

「文が上から下に実行されることと再帰呼び出しの関係」について, 補足する.

■通常関数呼び出しの場合 多少作為的だが, 次のようなソースコードがあったとする. 関数 helloworld を実行すると, hello と world が一行ずつ表示される.

```
Python3 1 def mynewline():
2     print("")
3
4 def myprint(msg): # メッセージを書いて改行する
5     print(msg, end="")
6     mynewline()
7
8 def helloworld():
9     myprint("hello")
10    myprint("world")
```

この実行過程の一部を図で模式的に表すと以下ようになる. C++ を含む多くのプログラミング言語では, 関数を呼ぶ (call) とその関数の実行状態や local 変数を管理する フレーム が作られる. 下図ではフレームを枠囲みで表した.



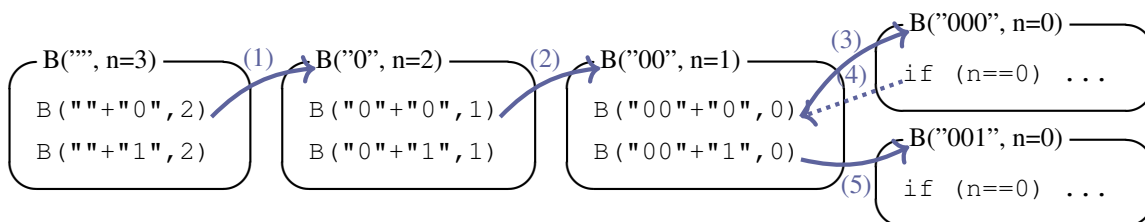
各フレーム内で見た時に, 文は書かれた順に実行される. たとえば, 一番左の helloworld のフレームにおいて myprint("hello") の実行が完了してから myprint("world") の実行が開始される. myprint("hello") の実行が完了するとは, myprint 関数の呼び出しが実行され (矢印 (1)), そのフレーム内で, 全ての文の実行を終えて制御が戻る (矢印 (4)) ことである. その過程では mynewline 関数も実行さ

れる (矢印 (2),(3)). 矢印 (3) や (4) のように, あるフレームで全ての文の実行を終えるか `return` が起こると, 呼び出し元の適切な位置に制御が戻される。

■再帰の場合 再帰の場合も同様に, 呼び出し毎にフレームが作られる。B.2.2 節の `m` 進数の例題を以下のように実装したとする。

```
Python3
1 def B(prefix, n):
2     if (n==0):
3         ...
4         return
5     B(prefix+"0", n-1)
6     B(prefix+"1", n-1)
```

この時、`B("", 3)` の呼出しは, 以下のように進行する。



ソースコード上は同じ関数であっても, 呼び出し毎に異なるフレームが作成されることに注意する。それぞれのフレームごとに, 引数, ローカル変数やどこまで実行したか, などの情報を維持する。このような情報の管理のために, **スタック領域**が用いられる (データ構造のスタックとは異なる)。実行が完了したフレームは破棄されるが, 完了していないフレームはメモリ上に維持する必要がある。そのため, 何段も関数を呼び出すと (たとえば 100 万), 実行時エラーになる場合がある。たとえば標準演習環境では, 8 メガバイトが限度である。

```
ssh0-01m:~ 0123456789$ ulimit -a
...
stack size          (kbytes, -s) 8192
...
1
2
3
4
```

B.3 整数型の理解

本資料では, 整数の表現として `int` と `long long` という型を必要に応じて使い分ける。両者では表現できる数の範囲に差がある。

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

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 B.1 のように, `int`

表 B.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
<code>int</code>	32	-2^{31}	$2^{31} - 1$	約 20 億
<code>long</code>	32/64			使用を勧めない
<code>long long</code>	64	-2^{63}	$2^{63} - 1$	C++11 から標準型, 以前は gcc 拡張
<code>__int128_t</code>	128			gcc 拡張, portable でないが強力

型や他の整数型で表現できる値の範囲には限りがある。

先程の例では,

$$1\,000\,000 \cdot 1\,000\,000 = 10^{12} > 2^{31} \approx 2 \cdot 10^9$$

と 32bit 整数で表現できる範囲を超える。つまり、このような場合は `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}

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++

```

1 #include <limits>
2 #include <iostream>
3 using namespace std;
4 int main() {
5     cout << sizeof(int) // バイト数
6         << '\n' << numeric_limits<int>::digits // 符号以外の bit 数
7         << '\n' << numeric_limits<int>::digits10 // 十進桁数
8         << '\n' << numeric_limits<int>::min()
9         << '\n' << numeric_limits<int>::max()
10        << endl;
11    cout << sizeof(long long)
12        << '\n' << numeric_limits<long long>::digits
13        << '\n' << numeric_limits<long long>::digits10
14        << '\n' << numeric_limits<long long>::min()
15        << '\n' << numeric_limits<long long>::max()
16        << endl;
17 }
```

```

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

なお表 B.1 には符号付き整数のみ記載した。符号なし整数(`unsigned int`, `unsigned long long` など)は、負の数を表現できない代わりに 2 倍程度の範囲の正の数を表現可能である。本資料ではほぼ使わない

^{*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。

が、必要に応じて各自把握のこと。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。

B.4 浮動小数と誤差

実数を扱う場合には、通常は浮動小数点数を用いる。

なお状況によっては、整数を、小数を表すために用いることが適切である。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)

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 (mantissa) を持つ実数表現 (浮動小数点表現) が利用される。

B.4.1 様々な誤差

浮動小数点数を使う場合には、誤差が含まれることを理解する必要がある。

誤差は必ずしも直感的ではない。例として、0.1 刻みで 0 から 1 まで処理するループを考える。この場合に適切な方法は、ループカウンタに整数を用いることである。一見等価なようにも 0.1 ずつ足す方法は予想と異なる結果をもたらす。

良い実装		0.0
Python3	1 <code>i = 0</code>	0.1
	2 <code>while i < 10:</code>	(中略)
	3 <code>print(i/10.0)</code>	0.8
	4 <code>i += 1</code>	0.9 # 全 10 行
悪い実装 (0.1 から...11 行表示)		0.0
Python3	1 <code>i = 0.0</code>	0.1
	2 <code>while i < 1.0:</code>	(中略)
	3 <code>print(i)</code>	0.7999999999999999
	4 <code>i += 0.1</code>	0.8999999999999999
		0.9999999999999999 # 全 11 行

このような現象は、次に説明する丸め誤差から生ずる。さらに、誤差を含む数同士の演算では誤差は拡大する。誤差の種類について、次の 4 種類の分類がある。

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)

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

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 = 0011111111010011001100110011001100110011001100110011001100110100
0.3 = 0011111111010011001100110011001100110011001100110011001100110021

```

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

B.4.2 double 型の理解

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:

$$(-1)^s (1 + m \cdot 2^{-52}) \cdot 2^{(e-1023)}. \quad (\text{B.2})$$

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++ 1 #include <iostream>
    2 #include <bitset>
    3 using namespace std;
    4 union double_long_long {
    5     double floating_value;
    6     unsigned long long integer_value;
    7 };
    8 void show_double(double v) {
    9     double_long_long u;
   10     u.floating_value = v;
   11     cout << v << " = " << bitset<64>(u.integer_value) << endl;
   12     long long sign = u.integer_value >> 63;
   13     long long exponent = (u.integer_value >> 52) & ((1<<11)-1);

```

```

14  long long mantissa = u.integer_value & ((1ul<<52)-1);
15  cout << "_sign_" << sign << endl;
16  cout << "_exponent_" << exponent << ' ' << bitset<11>(exponent) << endl;
17  cout << "_mantissa_" << bitset<52>(mantissa) << endl;
18  cout << "_=>" << (sign ? "-" : "+")
19       << "_(1+" << (double)mantissa/(1ul<<52)
20       << ")*_" << "2^(" << (exponent-1023) << ")" << endl;
21  }
22  int main() {
23      show_double(1.0);
24      show_double(-1.0);
25      show_double(0.5);
26      show_double(0.25);
27      show_double(0.75);
28  }

```

実行例: 式 B.2 と照らして確認せよ.

1 = 001111111111000000000000000000000000000000000000000 (略)	1
sign = 0	2
exponent = 1023 0111111111	3
mantissa = 00 (略)	4
=> + (1+0) * 2 ⁽⁰⁾	5
-1 = 101111111111000000000000000000000000000000000000000 (略)	6
sign = 1	7
exponent = 1023 0111111111	8
mantissa = 00 (略)	9
=> - (1+0) * 2 ⁽⁰⁾	10
0.5 = 001111111111000000000000000000000000000000000000000 (略)	11
sign = 0	12
exponent = 1022 0111111110	13
mantissa = 00 (略)	14
=> + (1+0) * 2 ⁽⁻¹⁾	15
0.25 = 001111111101000000000000000000000000000000000000000 (略)	16
sign = 0	17
exponent = 1021 0111111101	18
mantissa = 00 (略)	19
=> + (1+0) * 2 ⁽⁻²⁾	20
0.75 = 001111111101000000000000000000000000000000000000000 (略)	21
sign = 0	22
exponent = 1022 0111111110	23
mantissa = 1000000000000000000000000000000000 (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.

B.5 構造体 *struct*

データをまとめて管理する目的で、*struct* を使用する。以下のような文法で、整数の変数 *height* と *weight* を持つ、*Student* という新しい型を定義する:

```
C++ 1 struct Student {  
    2     int height, weight;  
    3 };
```

使用例:

```
C++ 1     Student a;  
    2     a.height = 150;  
    3     a.weight = 50;  
    4     cout << a.height << ' ' << a.weight << endl;  
    5  
    6     Student b = { 170, 70 };  
    7     cout << b.height << ' ' << b.weight << endl;  
    8  
    9     Student c = { 180 }; // weight == 0  
   10     Student d = { };   // height, weight == 0
```

- 各 *Student* 型の変数 (上記では *a*, *b*, *c*, *d*) の *height* や *weight* にアクセスするには、*.*(ドット) 演算子を用いる。
- 初期化は、波括弧 *{}* で行うことも出来る。

付録 C

Ruby

C.1 入出力

主に 2 章に関連したサンプルコードを紹介する。

C.1.1 ICPC Score Totalizer Software

Ruby

```
1 while true
2   judges = gets.to_i # 数字を入力
3   break if judges == 0
4   a = []
5   for i in 0..judges-1
6     a << gets.to_i
7   end # ここまでに配列 a に各審判のスコアが入っている
8   sum = 0
9   for i in 0..judges-1
10    sum = sum + a[i]
11  end
12  puts sum
13  puts a.max
14 end
```

C.1.2 列の操作

Ruby

```
1 n = 50
2 # 長さ n の配列を用意し各要素を 0 で初期化 (a[0] から a[n-1] まで有効)
3 a = Array.new(n, 0)
```

Ruby

```
1 a = Array.new(5)
2 a.each{|e| puts e}
```

Ruby

```

1 a = [3,5,1,2,4]
2 a.sort!
3 a.reverse! # a を逆順に並び替え
4 p a

```

Ruby

```

1 a = [0,1,2,3,4,5,6]
2 a[0,7] = a[3,4]+a[0,3]
3 p a # → [3, 4, 5, 6, 0, 1, 2]

```

C.1.3 Hanafuda Shuffle

Ruby

```

1 while line = gets
2   n, r = line.split("_").map{|i| i.to_i} # n,r 読み込み
3   break if n == 0
4   # n 枚の山を作る
5   # 作った山を表示してみよう
6   r.times {
7     p, c = gets.split("_").map{|i| i.to_i}
8     # シャッフル p,c を行う
9     # シャッフル毎に山全体を表示してみよう
10  }
11   # 山の先頭を出力
12 end

```

C.2 整列と貪欲法

主に 3 章に関連したサンプルコードを紹介する。

C.2.1 整列

Ruby

```

1 a = [3,5,1,2,4]
2 a.sort!
3 p a

```

C.2.2 Finding Minimum String

Ruby

```

1 N = gets.to_i
2 A = []
3 (1..N).each {
4   A << gets.chomp
5 }
6 ... // 整数の時と同様に A を整列 (sort) する
7 puts A[0]

```

C.2.3 ペアと整列

```
Ruby 1 a = [3,5]
      2 p a # [3,5] と表示
      3 b = [0.5,"X"]
      4 p b # [0.5,"X"] と表示
```

C.2.4 Princess's Marriage

```
Ruby 1 while line = gets
      2   pN, pM = line.split("_").map{|s| s.to_i}
      3   break if pN == 0
      4   pd = [] # <p,d>をしまう配列
      5   (1..pN).each { # pN回何かする
      6     d, p = gets.split("_").map{|s| s.to_i}
      7     pd << [p, d]
      8   }
      9   # pd を大きい順に並べてみよう
     10   # pd を出力してみよう
     11   # 並べ替えがうまく行っていたら答えを計算してみよう
     12 end
```

C.3 動的計画法

主に 4 章に関連したサンプルコードを紹介する。

C.3.1 Fibonacci 数列のナイーブな求め方

```
Ruby 1 def fib(n) # 遅いバージョン
      2   # p ["fib",n] 関数呼び出しの際に引数を表示
      3   if n==0
      4     0
      5   elsif n == 1
      6     1
      7   else
      8     fib(n-2)+fib(n-1)
      9   end
     10 end
```

C.3.2 平安京ウォーキング

入力例

```
Ruby
```

```

1 dataset = gets.to_i
2 dataset.times {
3   gx, gy = gets.split("_").map(&:to_i)
4   p [gx,gy]
5   matatabi = gets.to_i
6   # 問題文では変数 p を使っているが、表示コマンドの p と名前を分けた
7   (0..matatabi-1).each{|i|
8     x1,y1, x2,y2 = gets.split("_").map(&:to_i)
9     p [x1,y1, x2,y2]
10  }
11 }

```

C.4 基本データ構造

主に 6 章に関連したサンプルコードを紹介する。

C.4.1 文字列と読み込み

```

Ruby 1 word = gets.chomp # gets で一行読み込んで、chomp で行末の改行文字を取り去る
      2 puts word+word # 2 つつなげたものを表示する

```

C.4.2 スタック

Ruby では `unshift`, `shift` をそれぞれ `push`, `pop` として用いることにする。

```

Ruby 1 stack = [] # (配列を Stack として扱う)
      2 stack.unshift(3) # 要素を先頭に追加
      3 stack.unshift(4)
      4 stack.unshift(1)
      5 while stack.size > 0 # 要素がある間
      6   n = stack.shift # 先頭から取り出す
      7   p n
      8 end

```

Ruby では配列をキューとして扱うと便利である。

```

Ruby 1 Q = [] # (配列を Queue として扱う)
      2 Q << 3 # 要素を末尾に追加
      3 Q << 4
      4 Q << 1
      5 while Q.size > 0 # 要素がある間
      6   n = Q.shift # 先頭から取り出す
      7   p n
      8 end
      9 # 3, 4, 1 の順に表示される

```

C.4.3 優先度付きキュー

Ruby の場合は、今のところオンラインジャッジで使えるライブラリがなさそうなので、自分で二分ヒープなどを実装する必要がある。

```
Ruby
1 class PriorityQueue
2   def initialize
3     # 配列で二分木を実現する i 番目の要素の左の子供は 2i+1, 右の子供は 2i+2
4     # 親の要素は, 子供のどちらより優先度が高いとする
5     @array = []
6     @size = 0
7   end
8   def push(a)
9     # 最後に仮置きした後, 優先度が先祖より高ければ引き上げる
10    @array[@size] = a
11    heapify_up(@size)
12    @size += 1
13  end
14  def pop()
15    # 根の要素 (優先度最大) を取り出した後, 最後の要素を根に仮置きし, 調整
16    raise unless @size > 0
17    ans = @array[0]
18    @size -= 1
19    @array[0] = @array[@size]
20    heapify_down(0)
21    ans
22  end
23  def size
24    @size
25  end
26  def swap(p,q)
27    @array[p], @array[q] = @array[q], @array[p]
28  end
29  def equal_or_better(p,q)
30    (@array[p] <= @array[q]) <= 0
31  end
32  def heapify_up(n)
33    while n > 0
34      parent = (n-1)/2
35      break if equal_or_better(parent, n)
36      swap(parent,n)
37      n = parent
38    end
39  end
40  def heapify_down(n)
41    while true
42      l, r = n*2+1, n*2+2
```

```

43         break if @size <= 1
44         child = l
45         child = r if r < @size && equal_or_better(r, l)
46         break if equal_or_better(n, child)
47         swap(child, n)
48         n = child
49     end
50 end
51 end
52
53 Q = PriorityQueue.new
54 Q.push([50, 1]);
55 Q.push([20, 2]);
56 Q.push([30, 3]);
57 Q.push([10, 4]);
58 Q.push([80, 5]);
59
60 while Q.size() > 0
61     cur = Q.pop(); # 最小の要素を取り出す
62     p cur
63 end

```

C.4.4 文字列と分割・連結・反転

```

Ruby 1 word = "hello"
      2 s = word.split("")
      3 (0..s.length).each {|l|
      4     a = s.first(l)
      5     b = s.last(s.length-l)
      6     # p a
      7     # p b
      8     puts a.join("")+"_"+b.join("")
      9 }

```

```

Ruby 1 a = "hello"
      2 b = a.reverse
      3 puts b

```

C.4.5 集合

ruby の場合は Hash を用いると同様の機能を得られる。^{*1}挿入には `a.insert(element)` の代わりに `a[element]=1` を, 指定した要素の数・有無には `a.count(element)` の代わりに `a[element]` を, 集合の要素数は `a.size()` の代わりに `a.length` をそれぞれ用いる.

^{*1} 集合も用意されており `require 'set'` とすると使えるので, 興味のあるものは調べると良い.

Ruby

```

1 all = Hash.new(0)
2 puts all.length # 0
3 all["hello"] = 1
4 all["world"] = 1
5 all["good_morning"] = 1
6 all["world"] = 1
7 puts all["world"] # 1
8 puts all["hello!!!"] # 0
9 puts all.length # 3

```

C.4.6 連想配列

Ruby

```

1 table["hanako"] = 160
2 puts table["taro"] # 180
3 puts table["ichirou"] # 0 (*)

```

Ruby

```

1 table["ichirou"] += 1
2 puts table["ichirou"] # 1
3 table["ichirou"] += 1
4 puts table["ichirou"] # 2

```

Ruby

```

1 phone = Hash.new
2 phone["taro"] = 123;
3 phone["jiro"] = 456;
4 phone["saburo"] = 789;
5
6 phone.each{|key,value|
7   p [key, value]
8 }
9 # ["taro", 123]
10 # ["jiro", 456]
11 # ["saburo", 789]

```

Ruby の場合は、C++ より簡潔な記法が可能である。ループ中に使いたい変数を `|key,value|` の部分で指定する。

C.5 グラフ上の探索

主に 8 章に関連したサンプルコードを紹介する。

隣接リスト (例題 Graph) の読み込み

Ruby

```

1 N = gets.to_i
2 G = (1..N).map{ Array.new(N, 0) }
3 N.times{

```



```

4   u,k,*v = gets.split('_').map(&:to_i) # u,k は数, v は配列
5   v.each {|vi| # v の各要素 vi について
6       # (u-1) と (vi-1) をつなげる
7   }
8   }
9   G.each{|r|
10      puts r.join('_')
11  }

```

幅優先探索 (BFS) (8.2 章)

Ruby

```

1  Q = [0] # 始点 0 を入れたキュー (配列)
2  D = Array.new(N,-1)
3  D[0] = 0 # 始点への距離は 0, 他の距離は -1 としておく
4  while Q.size > 0
5      p ["debug", Q] # 各ステップでの Q の動作を確認 (後で消すこと)
6      cur = Q.shift
7      (0..N-1).each {|dst|
8          if ... then # cur から dst に移動可能で、dst が未訪問だったら
9              D[dst] = D[cur]+1
10             Q << dst # Q に dst を詰める
11         end
12     }
13 end
14 # D を表示

```

深さ優先探索 (DFS) (8.3 章)

Ruby

```

1  def dfs(src)
2      D[src] = $time # 訪問時刻を記録
3      $time += 1
4      (0..N-1).each {|dst|
5          if ... then # src から dst に移動可能で、dst が未訪問だったら
6              dfs(dst)
7          end
8      }
9      F[src] = $time # 訪問終了時刻を記録
10     $time += 1
11     # 関数の終わりで親に戻る (青矢印)
12 end
13
14
15 D = Array.new(N) # D[i] の初期値は nil
16 F = Array.new(N)
17 $time = 1
18
19 (0..N-1).each {|id| # 番号が若い節点から
20     if ! D[id] then # D[id] が未訪問だったら
21         dfs(id) # dfs を始める

```

```

22     end
23 }
24 # 出力

```

C.6 最短路

主に 9 章に関連したサンプルコードを紹介する。

BellmanFord 法 (9.3.2 節):

```

Ruby 1 V,E,r = gets.split(' ').map(&:to_i) # 入力
      2 Edge = []
      3 E.times{
      4   s,t,d = gets.split(' ').map(&:to_i)
      5   Edge << [s, t, d]
      6 }
      7 Inf = 10000*100000+100 # 全頂点をたどると最大
      8 C = Array.new(V,Inf) # 各頂点までの始点からの距離の上限
      9 C[r] = 0 # 始点
     10 V.times{ # V回
     11   count = 0
     12   Edge.each{|s,t,d| # 全ての辺 (s から t, コスト d) に対して
     13     if C[s] < Inf && C[t] > C[s]+d
     14       C[t] = ... # C[t] を更新
     15       count += 1
     16     end
     17   }
     18   break if count == 0 # 更新されなければ終了して良い
     19 }
     20
     21 V.times{|i|
     22   puts (C[i] == Inf) ? "INF" : C[i]
     23 }

```

C.7 数値積分

主に 15 章に関連したサンプルコードを紹介する。

C.7.1 Find the Outlier

```

Ruby 1 # 配列 $V に値が入っているとする
      2 def interpolate(n, e)
      3   sum = 0.0
      4   $V.each_index{|k|
      5     next if ... # k が n か E なら
      6     p = $V[k]

```

```

7      $V.each_index{|i|
8          p *= 1.0*(...-i)/(k-i) if ... # i が k でも n でも E でもなければ
9      }
10     sum += p
11 }
12 sum
13 end

```

Ruby

```

1  require 'scanf'
2  # 関数定義
3  while true
4      d = (scanf '%d')[0]
5      break if d == 0
6      \ $V = scanf('%f'*(d+3))
7      (0..d+2).each{|i|
8          if outlier(i) # i を無視すれば全て整合する
9              puts i
10             break
11         end
12     }
13 end

```

C.7.2 Intersection of Two Prisms

Ruby

```

1  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
2      w = []
3      polygon.each_index{|i|
4          p, q = polygon[i], polygon[(i+1)
5          # 辺 pq を考える
6          if p[0] == x
7              w << p[1]
8          elsif (p[0] < x && x < q[0]) || (p[0] > x && x > q[0])
9              x0, y0 = p[0], p[1]
10             x1, y1 = q[0], q[1]
11             x0, y0, x1, y1 = x1, y1, x0, y0 if p[0] > x
12             w << y0 + 1.0*(y1-y0)*(x-x0)/(x1-x0)
13         end
14     }
15     raise if w.length == 0
16     w.max - w.min
17 end

```

Ruby

```

1  def volume
2      $X.sort!
3      $X.uniq!
4      sum = 0.0

```

```
5   xmin = [($P1.min)[0], ($P2.min)[0]].max
6   xmax = [($P1.max)[0], ($P2.max)[0]].min
7   (0..$X.length-2).each{ |i|
8     a, b = $X[i], $X[i+1]
9     next unless xmin <= a && a <= xmax && xmin <= b && b <= xmax
10    m = (a+b)/2.0
11    va = width($P1, a)*width($P2, a)
12    vb = width($P1, b)*width($P2, b)
13    vm = width($P1, m)*width($P2, m)
14    area = .. // (a,va), (m,vm), (b,vb) を通る2次式の区間[a,b]の定積分
15    sum += area
16  }
17  sum
18 end
```

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, <https://book.mynavi.jp/ec/products/detail/id=22672>
- [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.kindaiakagaku.co.jp/information/kd0408.htm>
- [6] Bjarne Stroustrup, “The C++ Programming Language” (4th Edition), Addison-Wesley Professional, 2013

索引

-fsanitize=undefined, 9
-std=c++11, 9
-Wall, 9

Accepted, 7
ACM-ICPC, 6
AOJ, 6
arc, 74
ASCII, 132
assert, 181

back edge, 95
binary_search, 48
bitset, 69

complex, 116

degree, 74
deque, 63
diff, 13
Disjoint-set, 76

edge, 74

Fibonacci numbers, 191
forest, 75
front, 63

global, 131
gnuplot, 57

int32_t, 197
int64_t, 197

koeh, 56
Kruskal, 79

leaf, 75
long long, 196

map, 69
matplotlib, 58
memoization, 143
minimum spanning tree, 79

path, 74
POJ, 6
pop, 60
priority queue, 65
push, 60

queue, 63

recursion, 190
root, 75

Rutine Error, 13

set, 67
spanning tree, 79
stack, 60
struct, 200

tabling, 143
Time Limit Exceeded, 13
timeit, 37
top, 60
tree, 75
trunc, 135

ulimit, 195
Union Find, 76
unordered_map, 69
unordered_set, 67
unsigned, 196

vertex, 74

Wrong Answer, 13

エディタ, 8

親, 75

拡張ユークリッド互除法, 159
関節点, 101
緩和, 107

木, 75
逆元, 159
キュー, 60

クラスカル法, 79
グラフ, 74

子, 75
後退辺, 95
国内予選, 6
コッホ曲線, 56
コンパイルオプション, 9

再帰, 190
最小(重み)全域木, 79
最長共通部分列, 43

次数, 74

スタック, 60
スタック領域, 195

整数型, 196

節点, 74
全域木, 79
漸化式, 190

ターミナル, 8

頂点, 74

二部グラフ, 98
二分探索, 48

根, 75

葉, 75
橋, 101
パス, 74

フォルダ, 10
符号付き整数, 196
符号付き面積, 118
符号なし整数, 196
部分問題, 37
負閉路, 106
負辺, 106
フレーム, 194

辺, 74

末尾再帰, 192

メモ化, 143

森, 75

ユークリッドの互助法, 158
優先度付きキュー, 60

リダイレクション, 12
隣接行列, 90
隣接リスト, 90