Teach Yourself KMC

独習KMC vol.1

京大マイコンクラブ　著

巻頭言

田中 和彦

京大マイコンクラブ，初代会長，設立メンバの田中和彦です．

1977 年春に創立し，今年で 34 年目，米国のマイクロソフトよりは少々，短いですが，日本のマイクロソフトよりは長い歴史を持つ団体です．

その設立の経緯を紹介させて頂きます．

私は，小さいときから，父親の影響もあり，電気，無線に興味を持っていました．

小学校 6 年の時にアマチュア無線技師免許を取得し，自局開局の前に，大阪万博の記念アマチュア無線局で初交信しました．高校時代には，IC を組み合わせて測定器などを自作していました．

「マルチメータ」を作ってレーザと落体を組み合わせて重力加速度を測定し郷土，静岡の偉人の鈴木梅太郎賞を頂いたこともあります．

当時（今も）CQ 出版よりインタフェースという雑誌があり，そこに，非常に魅力的な記事が掲載されており，それは京都大学情報工学科の先生方による記事で，私は，京都大学情報工学科で学ぶことに憧れましたが，現役合格はならず，地元で一浪した後，念願の情報工学科に合格することが出来ました．

入学して，私は嬉しくて仕方がありませんでした．それは「これで本物のコンピュータが使える」という想いでした．私は，情報工学科の主任教授に会いに行きました．私は，ビットスライスチップを使って，複数の ALU などとマイクロプログラムで構成して日立の HITAC-10 相当のコンピュータを自作しようと考えていました．当時，NHK でコンピュータ講座が放映されておりそこで使われていたのが HITAC-10 だったのです．いきなりマイクロプログラムを作成する訳には行かないので，大学の大型計算機をつかってシミュレーションをしようと思い，主任教授に「コンピュータを使わせて下さい」と直談判に行ったのです．

主任教授の答えは「あのな，君らはな，学部生になれば（いやでも）実習はあるし，卒業したらそういう仕事に就く」「今，やらんでもええんとちゃうか」「今は，学生として，アマチュアとしてしか出来んことをしたらどうや」でした．

今，思えば，主任教授は，訳の分からない学生を上手くあしらっただけかもしれません

が，私は，ガッカリした半分，なるほどの思い半分で，それならば，仲間を集めてアマチュアならではのことをして見ようと思いました．

そこで，仲間達に声をかけて，教養学部校内の，正面玄関の 2 階，確か A215 教室だったと思いますが，そこに，毎週木曜日に集まることにしました．

会の名称は，Computer Evolution Group (CEG) で，これは，私の，コンピュータは革命 (Revolution) 的に変化するのではなく，生物の進化 (Evolution) のように進化するのであり，その進化に関わる，という思いで命名しましたが，メンバの「『京大マイコンクラブ』の方がええんとちゃうの」の一言で改名しました．

当時は，4bit あるいは 8bit の CPU，というか MPU が市販され始めたころで，関西の某社が，インベーダーゲームのために Z-80 を量産し一気に 850 円という今でも驚きの価格になるまではなかなか価格が高くて学生には手が出ない状況でした．

入出力装置も，今，普通に使われているタイプライター様のキーボードは数万円と非常に高かったので，表示は LED ランプ，入力は 1bit ずつ入力するトグルスイッチで，フルキーボードを入手したらそれはもう自慢ものでした．

このような状況ですので，毎週，A215 教室に集まって何をしていたかというと「モトローラから新しい MPU が出たらしい」「インテルからは」「いやこっちのアーキテクチャの方が美しい」など，全くマシンは無いのに「あーだ」「こーだ」と「マイコンクラブ」と言うよりは「無いコンクラブ」の様相でした．

やがて，自作マシンやら，Apple-II などを各人が所有するようになりました．

そこで部室をという話になりましたが，大学構内では，アマチュア無線部が無線機を盗まれたという噂もあり，学外で場所を確保することになり，確かうどん屋さんの 2 階を借りて部室がスタートしました．

その後，8bit マシン，例えば，PC-8001，MZ-80，Apple-II，FM-8，自作機を 4800bps の 20mA 電流ループで接続するローカルネットワーク PLANET の開発などが行われ，池袋のサンシャインで，ASCII 出版の後援で，PLANET を使ったデモンストレーションを行ったりしました．

大阪で開催された SF 大会「ダイコン」で複数台の 8bit PC で宇宙船「エリトザハース」のシミュレーションを行ったり，ネットワーク経由のテニス，ガンダムの対戦ゲームなど，今でも通用するのではないかと思える内容でした．

その後の現在に至る活動は，正直，初代会長の私も，余りにも多くのことが活発に行われていて，把握しきれていないほどです．

部室のマシンを初期のインターネットに接続したり，コンピュータグラフィックスやプログラミングコンテストで名を馳せたりと，「中興の祖」が何人も出現しました．

創部の頃は，パソコン，というか，マイコンは非常に特殊な物でした．

34 年後の今では，マイコンどころかパソコンも死語かもしれません．

しかし，今でも，何かをやろう，集まって皆でやろう，という意志を持って活動し，京大マイコンクラブが存続していることは，創部の頃の世界はもはや化石や神話ではないか

と思っていますが，大変に嬉しく，また，将来への希望を感じます．

私が，入学当初，意気揚々に，学科主任教授に面会し「アマチュアならでは」と言われて感じたものを引き継いで頂ければ嬉しいです．

今日，言うまでもなくテクノロジーはもの凄く進化しています．

創部の頃の CPU のクロックは数 MHz でしたが，今ではポータブルデバイスでも GHz は当たり前です．

テクノロジーが高度であるほど使うことは簡単です．例えばスマートフォンは誰でも簡単に使えますし，そうでなければ意味がありません．一方，そのテクノロジーを創り出す人々が必要です．単に便利に使うのではなく，何か新しいこと，新しい使い方，もっと皆がハッピーになること，それらを創造することに，若い皆さんの関心，エネルギー，パッション，が向いて貰えたら良いな，と思っています．

様々なものがサイバー空間に置かれ，仮想化しています．ある意味で現実感が失われています．一方，先日，部室を訪問し，驚いたのですが，外観はオンボロのアパートのドアが自作の電子カードシステムだったり，よさげなコタツに入ってプロジェクターでプレゼン資料を映しまったりと議論できる環境だったりと，日々，ビジネスに追われる身からすれば，通常のビジネスではあり得ない環境で，羨ましい限りでした．このような環境を大事にしてください．

さらにこのような感覚を KMC の外の皆さんと共有出来たらもっとすばらしいと考えています．

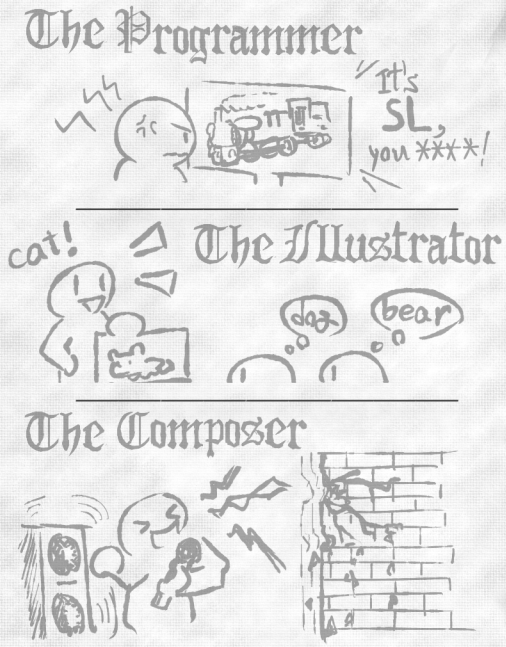
KMC 創業メンバの私達も頑張ります，皆さんも力を発揮して未来を創造して下さい！

情報通信総合研究所
取締役・主席研究員
田中和彦

目次

Page 7

初心者向けコーナー



競技プログラミング入門

kirita

競技プログラミングとは

ICPC などのコンテストで競われるのはプログラミングの技能です——といってもゲームを作ったりソフトを開発したり，ということはありません．例えばこういった問題が出題されます：

> 2 個の文字列が与えられたとき, 両方の文字列に含まれる文字列のうち最も長いものを探し, その長さを答えるプログラムを作成せよ.
>
> 日本情報オリンピック第 7 回本選より

入力の例として 2 つの文字列が “ABRACADABRA” と “ECADADABRBCRDARA” だった場合 “ADABR” という文字列が求める最長のものになります． もっとも，実はこれだけでは問題としては成立しません．プログラミングコンテストで出題される問題には次のような制限が問題文に書かれています．

> 文字列は英大文字からなり, 各々の文字列の長さは 1 以上 4000 以下である.
>
> 同

さて，このような問題をどのように解けばいいのでしょうか？ここではとりあえず解法だけを考えてみたいと思います．

まず思いつくのは，一方の文字列から部分文字列を全て列挙し，もう一方の文字列に出現するもののうち最長のものを答えとするやり方です．これは，正解です．なにしろ答えの候補となるものを全てしらみつぶしに探しているのだから間違いようがありません．

しかし，文字列の長さが最大で 4000 もあることを考えるとこのやり方では少し効率が悪そうな気がします．当然人間がやるとかなりの時間がかかってしまいます．では，コンピュータがやるとどうなるのか？少し計算してみましょう．

しらみつぶし解法の計算量について

2つの文字列の長さをそれぞれ n, m とします．まず長さ n の文字列について長さが s の部分文字列は $n-s+1$ 個存在します．また，長さが s である2つの文字列が同一であるかどうかを判定するには個々の文字を調べていく必要があり，最大で s 回の比較をすることになります．よって，1つ目の文字列から取り出した部分文字列がもう一方の文字列に出現するかどうかを判定するには $s(m-s+1)$ 回の比較が必要であることがわかります．s は1から $\min(n,m)$（$=N$ とおく）までの値をとりうることから，全体で

$$\sum_{s=1}^{N} s(m-s+1)(n-s+1) \geq \frac{N(N+1)(N^2+11N+6)}{12}$$

だけの回数比較することが可能性としてあります．普通こういったとき，目安として次数が最大の N^4 をオーダーといい，$O(N^4)$ などと書きます．

さて，文字列の長さは最大で 4000 でした．このとき $N^4 = 256 * 10^{12} \fallingdotseq 10^{14}$ です．実際このくらいにもなるといくらコンピュータとはいえ処理には現実的でない時間がかかってしまいます．さらに大半のコンテストには実行時間に制限が設けられており，時間内に処理できるのは 10^8 が限度です．この問題では，大体 $O(N^2)$ くらいのオーダーで答えを求められればよさそうです．

高速に解く方法

両方の文字列の最初の位置から始めて順番に文字を比較していくと，例えば “ABCBA” と “AACBB” ならば “oxoox”（同じならば o，違えば x という文字列）という結果が得られ，この場合の両者に出現する部分文字列は長さ 2 が最長であることが分かります．実は，この判定をどちらかの文字列の開始位置をずらして行うことによって全ての場合を調べることができます．これならばずらし方の総数が $n+m$，1 回の判定にかかる回数が $\min(n,m)$ なのであわせて $(n+m)\ \min(n,m)$ 回の比較で答えが出せます．このオーダーは $O(N^2)$ ですからさきほど述べたことより，コンピュータならば一瞬で解を求めることができます．

この記事で書くこと

このように問題の解法を考えると，次は実際にプログラムのソースコードを書きます．そして書いたコードを……どうするのでしょうか？普通人がコードを読んでそれが正しいかどうかを判定することはありません．問題の入力をいくつかプログラムに実行させて，それが審判する人の作成したプログラムと出力内容が同じであればそのプログラムは正しい，ということになり，これを Accept されたなどと言います．この判定にはいくつかの方法があり，入力をダウンロードして手元でプログラムを実行し出力をアップロードした

り，ソースコードをサーバに提出すればサーバ側で実行して判定してくれたりということもあります．この記事では，おもにそういったジャッジサイトや，各種のコンテストについて紹介していきたいと思います．

ジャッジサイト

ジャッジサイトとは先ほども述べたとおりソースコードを提出するとそれが正しいかどうかを判定してくれるサービスです．主に後述する ACM-ICPC やオリジナルのコンテストで出題された問題が数多く収録されています．

先ほどの情報オリンピックの問題は Aizu Online Judge[*1]（以下 AOJ）に収録されています．AOJ に限らずこの種のサイトを利用するためにはまずアカウントを取得する必要があります．大体トップページに登録ページへのリンクがあるのでそこを見ます．[*2] どのサイトでも必須なのが ID とパスワードです．他にメールアドレスや所属などを聞かれることもあります．無事登録できたら次に問題を探します．AOJ の場合は問題がいくつかの volume に分かれており，特に Volume100 が初心者向けの問題セットになっています．先ほどの情報オリンピックの問題は問題番号 0528[*3]にあります．この記事では実際のコードの書き方といったようなことは取り扱いません．書店に行けばその類の本はたくさんあると思いますのでそちらを参照してください．

さて，コードが書けたとします．コードを提出するにはページ右上にあるアイコンをクリックすると submission form というのが出てきます．言語を選び，そこの source code に自分が書いたコードをコピーして下の submit というボタンをクリックします．ただし，これはあくまで執筆時点 (2011/6/28) の情報です．一般にウェブサービスは変更がよく行われるので，これを読まれている時にはページ内容が変わっているかもしれません．その点についてはご了承ください．

コードを提出すると judge status のようなページにリダイレクトされると思います．ここで表示される Status については AOJ のページ[*4]に詳しくのっていますがここでも少し説明してみます．

Compile Error (CE)
: コンパイルに失敗したことを意味し，大体の場合文法ミスなどが原因です．必要なヘッダファイルがインクルードされていないということもありえます．

Runtime Error (RE)
: 実行時エラーが起きたときに表示されます．配列の範囲外参照，0 除算などが主ですが，main 関数が 0 を返さないときにも RE になるようです．

[*1] http://judge.u-aizu.ac.jp/onlinejudge/
[*2] AOJ の場合は http://judge.u-aizu.ac.jp/onlinejudge/register.jsp
[*3] http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0528
[*4] http://judge.u-aizu.ac.jp/onlinejudge/status_note.jsp

Time Limit Exceeded (TLE)
: プログラムの実行時間が指定されているタイムリミットを越えたときに表示されます．たいていのジャッジサイトではそれを越えた時点でプログラムの実行が打ち切られますが，AOJ では少なくとも 10 秒は実行するようです．また，Java のときは指定されている時間よりも 2 倍長くとられるようです．

Memory Limit Exceeded (MLE)
: プログラムで確保したメモリが制限を越えると表示されます．あまり気にしなくてもいいかもしれません．

Wrong Answer (WA)
: プログラムの出力が答えと違っているときに出ます．プログラムのバグ，アルゴリズムの間違い，コーナーケース（落とし穴的な入力）など，様々な原因が考えられます．WA の 1 つに Presentation Error (PE) というのがありますが，これは答えは合っているが出力の形式がおかしい，たとえばスペースが 1 個多いとか改行が必要であるとかいったことです．問題文を読めば容易に解決するでしょう．

Accepted (AC)
: プログラムの出力がジャッジ側と一致したことを意味します．つまり，そのプログラムが正解であるという事です．

ここでは AOJ を例に説明しましたが，ジャッジサイトは他にもたくさんあります．記事の後半ではそれらについて触れていきたいと思います．

Aizu Online Judge（略称 AOJ または AIZU）

既に取り上げましたが，Aizu Online Judge は日本の会津大学が運営しているジャッジサイトです．ICPC の国内および地区予選，日本情報オリンピック，パソコン甲子園（会津大学などが開催している高校生向けの大会）などの日本で開催されているコンテストの問題が多く収録されており，日本語の問題文もあります．ジャッジサイトはたいてい海外のものであることがほとんどであることを考えれば，日本語の問題文が読めるというのはかなりの利点でしょう．また，他にはない特色として API が提供されていることが挙げられます．

Peking University Online Judge（略称 PKU または POJ）

PKU[*5]は中国の北京大学が運営しているサイトです．このサイトは何と言っても収録されている問題数が膨大であり，執筆時点でも 3000 問弱はあります．国内外の ICPC や，各種のコンテストの問題が幅広く収録されています．問題文はほとんどが英語で，たまに中国語がある程度です．

[*5] http://poj.org/

UVa Online Judge（略称 UVa）

UVa[*6]はスペインのヴァリャドリッド大学が運営しているサイトです．このサイトには通常の問題セット以外に ACM-ICPC Live Archive というものがあり，これは世界各地で開催された ICPC の国内，地区予選および World Final の問題が多く収録されています．過去問の練習をしたいときなどにうってつけでしょう．

Project Euler

Project Euler[*7]は実はジャッジサイトではありません．ここではたとえば次のような，数学的で簡潔な問題が出題されます．

> 正整数 n に対して $f(n)$ を n の倍数の中で各桁がすべて 2 以下の整数であるようなもののうち最小の値と定める．$\displaystyle\sum_{n=1}^{10000} \frac{f(n)}{n}$ を求めよ．
>
> Project Euler 問題 303 Multiples with small digits より抜粋．訳は筆者

そしてページには答えのみを入力させる欄が 1 つだけあります．純粋に答えだけで判定されるので，たとえば紙と鉛筆だけで正解をもらうことも可能ですし，1 日中パソコンでプログラムを実行しておくというようなこともできます．数学が好きだという人はぜひ挑戦してみてはいかがでしょうか．

コンテスト

ここまでで主なジャッジサイトを 3 つ紹介しました．では次にコンテストを紹介しましょう．

ACM-ICPC

今までにも何度か言及してきましたが，ICPC (International Collegiate Programming Contest) とは，その名の通り国際的な大学対抗のプログラミングコンテストであり，長い歴史を持ちます．大学対抗と銘打ってありますが基本的に 1 つの大学から何チームでも出場できます．各チームは 3 人のメンバーとコーチが 1 人必要です．具体的な参加資格などは公式サイトを参照してください．

ICPC は国内予選，地区予選，世界大会 (World Final) の順に進んでいきます．目安としては，国内予選は大体 6 月の終わりごろ，地区予選は 12 月の頭あたり，そして世界大

[*6] http://uva.onlinejudge.org/index.php
[*7] http://projecteuler.net/

会は 2 月ごろと，1 年をかけて開催され，毎年白熱した戦いが繰り広げられます．

情報オリンピック

情報オリンピックは中高生を対象としたプログラミングの大会です．まず日本情報オリンピックがあり，予選，本選を通過すると合宿への参加資格が得られます．その合宿で選抜された上位 4 名までが国際情報オリンピックに出場できます．日本情報オリンピックは JOI (Japan Olympiad in Informatics)，国際情報オリンピックは IOI (International Olympiad in Informatics) と呼ばれています．IOI における日本人選手の活躍はめざましく，2010 年度のカナダ大会では金メダル 2 個，銀メダル 2 個を獲得しています．

ACM-ICPC ルールとは

ここで一般に ACM-ICPC ルールと呼ばれているルールについて説明しましょう．このルールは ICPC で採用されているものですが，他のコンテストでもしばしば採用されることがあります．次のような方法で順位を決定します．

- まず，解いた問題数が多いほど順位が上になります．
- 解いた問題数が同じならば，次に述べるペナルティが少ないほど順位が上になります．
- 解いた問題について，解くまでにかかった時間＋解くまでに出した不正解の数×20 分（解けなかった問題についてはペナルティは発生しない）．

一方，この ACM-ICPC ルール以外にももう 1 つ，競技プログラミングの大きな位置を占めている，まさに「競技」プログラミングと呼ぶべきルールがあります．そしてそれを採用しているのが TopCoder[*8]や Codeforces[*9]などのコンテストです．まずは TopCoder から紹介していきましょう．

TopCoder SRM (Single Round Match)

TopCoder を始めるにはまずアリーナというアプレットを公式サイトからダウンロードしてくる必要があります．コンテストなどは全てこのアプレット上で行われます．また，便利な環境を整えるためにプラグインを導入することもできます．このあたりはやや面倒な作業ですが，公式サイトや，あるいは解説ページを探して参考にしながら進めるとよいでしょう．

TopCoder にはいくつかの部門がありますが，その中で競技プログラミングにもっとも近いのが SRM です．SRM は月に 2, 3 回程度開催されます．日程は公式サイトにのって

[*8] http://www.topcoder.com/tc
[*9] http://www.codeforces.com/

いますが，はてなの topcoder 部[*10]を見ると日本時間が分かりますし，他のコンテストの情報もあるので非常に便利です．

SRM ではコンテストに参加するとその成績に応じてレーティングというものがつきます．レーティングは次のように色分けされています．

0–899	900–1199	1200–1499	1500–2199	2200–
灰色	緑色	青色	黄色	赤色

また，レーティングが 1199 以下だと Div2，1200 以上だと Div1 のコンテストにそれぞれ割り当てられています．Div2 と Div1 では問題の難易度が異なります．初回のレーティングは 1200 ですが Div2 のコンテストに割り当てられます．

SRM のルール

SRM では 3 問のセットを 75 分で解きます．問題は難易度順に Easy, Medium, Hard となっていて，それぞれにスコアが設定されており，250, 500, 1000 が標準的なスコアです．

コンテストが始まるとまず問題が提示されますが，始まったばかりのときは全て「開かれていない」状態になっています．開かれていない状態の問題は見ることができません．これを開くと問題文を見たり，ソースコードを提出することができますが，その問題を解くことによって得られる点数は開いてからの時間によって決まります．なるべく早くソースコードを提出することが高得点につながります．そして，ここが一番の特徴と言えるのですが，提出したソースコードはコンテストの終わりまで正解かどうか判定されません．ではいつ判定されるのか？ひとまず 75 分が過ぎると提出できなくなります．この始めの 75 分が coding phase です．少し休憩を挟んでから次に challenge phase というのが 15 分間あります．このフェーズでは他人のソースコードを読み，プログラムのバグやアルゴリズムのミスを見つけ，間違いを誘発するようなテストケースを送ることができ，それが実際に間違いであればそのコードを「撃墜」することができます．撃墜に成功するとポイントがもらえますが，失敗するとマイナスのポイントとなってしまいます．また参加者はコンテストの開始直前にルームに各 20 人ほど割り当てられます．チャレンジは同じ部屋にいる人のコードに対してしかできません．

challenge phase が終わると最終的な system test に入ります．これをパスするとようやく点が確定し，チャレンジも含めたポイントの合計によって順位が決定します．

[*10] http://topcoder.g.hatena.ne.jp/

SRM その他

SRM が他のコンテストと異なってる点として，答えを求める関数を書くだけでよい，ということがあります．普通は標準入出力などから入力を読み込み，答えを出力しますが，入出力の部分のコードを書かなくてよいので少し楽かもしれません．が，実質的にはそこまで変わることはないでしょう．アリーナでは過去の SRM の問題を解くことができます．初めてでまだなにも分からないときには過去問をみて様子をつかんでみるとよいでしょう．

Codeforces

SRM と同様のコンテストサイトが Codeforces です．Codeforces は比較的新しく，division の境目やレーティングに対応する色などが時々変わっていたりします．SRM は全てアリーナでコンテストなどを行いますが，Codeforces はウェブブラウザから参加できます．問題セットは概ね 5 問を 2 時間で解く形式です．普通のラウンド以外にも時々 Unknown language contest といって使える言語がコンテストが始まるまで分からないなど実験的ともいえるラウンドがあったりします．また，SRM では coding phase が終わった後に challenge phase が始まりますが，Codeforces ではコードを書いている最中に，他人のコードを見て Hack（SRM でいう challenge）することができます．入出力はほぼ標準入出力からであり ICPC 風のコードを書くことになるため，こちらのほうがとっつきやすいという人もいるかもしれません．

T シャツを手に入れよう！

5, 6 月には次のような流れで進められるコンテストがいくつか開催されます．

Qualification Round
: 出場資格（年齢制限など）を満たしていれば誰でも参加できるオンラインラウンドです．通過する人数はやや多目にとられます．

Online Round
: これもオンラインで何度か行われ通過者を絞っていきます．ある程度上位まで進めるとどのコンテストでも T シャツがもらえます．

Onsite Round
: 実際の会場で行われる決勝です．今までのラウンドを勝ち抜いてきた強者たちが競うため例年ハイレベルな戦いが繰り広げられます．

ともすれば目的を見失ってしまいそうな日々のコンテスト生活において，T シャツ（さらに上位に進めば賞金も！）という現物的な目標は非常に励みになります．もちろ

んこのような大会で上位にランクインできることはこの上ない誇りでしょう．以下では TopCoder Open Algorithm 部門と Google Code Jam の 2 つを紹介します．

TopCoder Open Algorighm 部門（略称 TCO Algo）

TCO Algo は TopCoder によって開催されている大会で，前述した SRM と同じくアリーナ上で行われます．3 回の QualificationRound によって次に進める 2000 人が選ばれ，その後の 5 回に及ぶ Online Round によって決勝の出場者が決定されます．2010 年度の TCO では Algorithm 部門と Marathon 部門の 2 つの部門において日本人が優勝しました[*11]．

Google Code Jam（略称 GCJ）

Google が 2008 年から毎年開催しているコンテストです．GCJ はプログラムの実行をコンテストのサーバ側で行うのではなく，テストケースをダウンロードして手元で実行する形式です．また，1 つの問題に対してテストケースが Small と Large の 2 つあり，それぞれに点数が設定されています．これによって遅い解法でも部分点を狙うことができるシステムになっています．

書籍や参考となるサイト

最後に，競技プログラミングを勉強する上で参考となるものをいくつか挙げてみます．

プログラミングコンテストチャレンジブック[*12]

問題集のような形式でページを進めつつ競技プログラミングにおいて必要なアルゴリズムのほとんどを網羅している本です．解説が分かりやすく，掲載されているソースコードもよく考えられたものになっています．競技プログラミングを志す人ならば必携の 1 冊と言えるでしょう．

deq notes[*13]

主に ICPC の問題を解説しているサイトですが，それだけでなく，初心者がつまづきやすい入出力の書き方や幾何問題の解き方についての記事，典型問題についての解説などが分かりやすく書いてあり，非常に参考になります．

[*11] http://plusd.itmedia.co.jp/pcuser/articles/1010/19/news032.html
[*12] http://book.mycom.co.jp/book/978-4-8399-3199-5/978-4-8399-3199-5.shtml
[*13] http://www.deqnotes.net/acmicpc/

Spaghetti Source[*14]

各種アルゴリズムの C++ での実装をまとめたサイトです．量がかなりあり，解説は簡潔ですがそのままコピーして使うこともできます．

どうしても問題が解けないとき

コンテストに参加したりジャッジサイトで問題を解いている人の中には，自分のブログにコンテストの参加記録や自分が解いた問題の解説を書いたりしている人もいます．もし自分が取り組んでいる問題がどうしても解けないと思ったら，ネットを検索してそのような解説記事を探してみてもいいかもしれません．また，TopCoder や Codeforces ではコンテスト後に Editorial といって問題の解説をした記事をサイトに載せることもあるので，それを読んでみるのもよいでしょう．

終わりに

この記事では競技プログラミングについて，たくさんのサイトやコンテストなどを紹介してきました．競技における醍醐味は，なんといっても AC がもらえた（問題が解けた）ときの達成感です．筆者も WA や TLE に悩まされながら，日々 AC を追い求めています．これを読んだあなたも，是非コンテストにチャレンジしてみてはいかがでしょうか？

[*14] http://www.prefield.com/algorithm/

イラストを設計しよう

Moko

どうしてパッとしない絵になってしまうのだろう……

コンピュータを使う方の中には，イラストを描く方もたくさんおられると思います．でも，いくら描いてもいまいちパッとしない，pixiv にアップロードしてみてもなかなかコメントや評価が付かない，ということはありませんか？一目見たときに「お！」と思えるイラストは，一体どのようにして描かれているのでしょうか．

全ては「画力」によるなどと片付けられてしまいがちですが，私は「画力」という言葉はバズワードだと思っています[*1]．目を引くイラストとは，「状況やテーマが（時にしつこいほど）明確なイラスト」です[*2]．そこに，「それっぽく描く力」が加われば百人力になります．

では，そういった「明確な」イラストを描くためにはどうすればいいのでしょうか．それには上達を促すための守るべきルールと，方法論とがあります．まずは守るべきルールからお話しします．

4 つの「守るべきルール」

イラストは何も考えずに描くと，つまらない・しまりのないものになりがちです[*3]．そんなものにしないために，4 つの守るべきルールがあります．

1. 完成させる

pixiv その他のグラフィッカーがたくさんいるコミュニティでは，ラフや下書きをアップロードしたものが多く見られます．単に楽しく描いているだけならよいのですが，もし

[*1] そのわりには「画力くれ」という表現は巷でよく聞かれますね．

[*2] プロのイラストは状況や雰囲気がしつこいくらい明確になっているはずです．だからこそキャラクターに感情移入できたり，懐かしみを覚えたりできるのです．

[*3] プロがインスピレーションのままに作った作品でも素晴らしい物がある，と思われる方もいるかもしれませんが，これについては後述します．

上達したいならこれはやってはいけません．

あなたのラフや下書きを見てください．たくさんの線でごちゃごちゃと描きこまれていませんか？人間の脳は，こうした線がたくさん入った絵をみると都合よく処理して，都合よくよい感じの絵に認識してしまいます．実際，下書きではよい感じだったのに，清書してみるとイマイチだった……という体験をした人は多いはずです．それはこの，脳の錯覚によるものです．ラフの段階からすっきりと 1 本の線で描くことができないなら，ラフのまま置いておくことは絶対にしてはいけません．必ず線を整理し，1 本の線で綺麗に描くところまで持って行きましょう[*4]．

そしてきっちりと線をひいたら，きちっと塗って，背景も付けて，多少粗くてもよいので「描きかけ」と思われないところまでやりましょう．やりかけで放置すると絶対に上達しません．完成させるまでのプロセスで学び取れることや試行錯誤することは数多くあり，これを体験しないまま次の題材に行くのは意味がありません[*5]．

2. 計画する

PowerPoint 等のプレゼンテーションを作る際にも言われること[*6]ですが，いきなりコンピュータに向かってモノを作るのは非常に効率が悪いこととされています．何故ならコンピュータは，「ただのイラストを描くための道具」だからです．

計画無しに木材とインパクトドライバーに向かっても何もできないのと同じように，計画無しに Photoshop や SAI の画面を見ていても何もできません．コンピュータに向かう前にやるべきことがあります．

たとえ非生産的でも，コンピュータに向かっているのが死ぬほど好きだと言うなら止めはしません．しかし，あらかじめ「どんな目的で」「何を」「どんなアングルで」「どんなシチュエーションで」「どんな線で」「どんな色で」「どんな塗り方で」描いて，「どう仕上げるのか」まで決めてあるほうが，絶対に迷いも少ないし，速くよいものが仕上がります．イラストを描いていて詰まったときほど，時間が無駄に思える時はありません．

ただし，ゴールが見えていて，そこに行くまでの道筋を模索する（例えばいろんなフィルタや色合いを試してみるなど）のに時間を費やすのは無駄ではありません．要するに目的もなくだらだら描くのは良くないのです．計画を立てましょう．

3. 資料を探す

描きたいものが決まっているのなら，「それがどんなものか」「どうやって描けばいいのか」がわかっている方が効率よく，正確に描けます．その際に大切なのが資料集めです．

[*4] 厚塗りのように線画の重要性が低い描き方もありますが，それでも私はちゃんと線を描くことをおすすめします．これについても後述します．

[*5] KMC のお絵描き勉強会「イラスト品評会」では未完成の作品を持ってきた人は蜜柑星人と呼ばれ，夏は甘夏みかん（大），冬は普通のみかん（小）を投げつけられる運命が待っています．

[*6] 『プレゼンテーション zen デザイン』，ガー・レイノルズ著，ピアソン・エデュケーションより

例えば銃火器・刀などの武器，正装や袴，軍服，制服などのふつうの服とは異なる服装[*7]，図書館や遊園地など特定の施設の写真などは，思いつきでパッと描けるものではありません．こんなものを資料無しにだらだら描いていても時間の無駄ですし，うろ覚えで描くと下手くそに見えがちです．昔は図書館を使ったり取材したりせざるを得ませんでしたが，今はコンピュータで世界中の情報を引っ張ってこれます．必ず資料集めをしましょう．

4. 作品は保管する

作品を捨てるのは，上達を妨げる罪深い行為です．絵がなかなかうまくならない内は，描いても描いても満足の行くものがなかなかできません．デッサンが狂っている，バランスがおかしい……．うまく描けたかな？と思っても，数日経って見てみるとひどい出来の絵にしか見えない……．何か思い出がある方も多いことでしょう．私も思い出ぽろぽろどころじゃ済まない状態です．7 年前の描き始めのころのイラストなどを見返すとそれこそベランダから飛び降りたくなります．……こんな経験は，特に描き始めでよくするものですが，「とても見ていられないから」とイラストを捨てる・データを消すことは絶対にしないでください[*8]．

これまで作ったものは，なんであれあなたの一部です．ひどい出来であったなら，折に触れそれを見て反省することが大切です．

反省のできない絵描きは上達できません．どこが悪かったか・どこがうまく描けたかのフィードバックがないのですから……．

他にも，作品を取っておけば，アイデアに詰まったときに昔の作品から引っ張ってくる，といったこともできます．あの時うまく描けなかったけど今なら，というリベンジも出来ます．今回例に取り上げた絵も，少なからず以前描いたイラストから何かしら引っ張ってきています．

イラストを設計しよう

次に，明確なイラストを描くための方法論のお話をします．この冊子の中表紙「最近の KMC」を例に取りましょう．

紙と鉛筆で始める

先述したとおり，コンピュータは「作るための道具」です．図面は紙と鉛筆，もしくはそれに準ずるもので始めましょう．コンピュータでやってもいいのですが，その場合は余計な情報を与えないペイントやメモ帳など，簡素なツールがおすすめです．繊細な線が描

[*7]「普通の服装」でもファッション誌などを参考にすべきです．特に資料がないまま流行の服を描こうとすると 9 割方残念なことになります．

[*8]同勉強会では，部室に「お絵描き」というボックスをつくり，そこにあらゆる絵を溜め込んでいます．最初のうちはみなさん「黒歴史を入れたくない」などとのたまうのですが，強制なのでそのうち慣れていきます．

けたり，色々できるツールを目の前にすると，全体よりも細部に目が行ってしまいがちです．線の色とかちょっとしたズレなんて，考える段階ではどうでもいいですよね．

■テーマを決める　今回のように「なんのためのイラストか」が決まっている場合は簡単ですが，そうでない場合はまずイラストのテーマを決定します．描きたいイラストをもっともよく表せる短いセンテンスや単語を探します．今回は「最近の KMC」という記事にふさわしいイラストですから，「これまで活動してきた足跡」と「アクティブな感じ」をテーマにしました．まずはこの 2 つのテーマを紙に書いて，そこから考え始めます．

■描くものを決める　次に描くものを決めます．もしかすると，こちらから先に決める人が多いかもしれません．今回は KMC の部誌ですから，KMC メンバーが考えたオリジナルキャラクターを使います．アクティブで走り回っていそうなものといえば金色の巨大ニワトリ「ふぇにくす」で，ふぇにくすと常にセットなのは「みこたん」です．そこで，この 2 つを使ったイラストを考えます．

図 1　今春の新勧用看板．みこたんとふぇにくすが使われている．

■シチュエーションを決める　テーマから，イラストのシチュエーションを考えます．「走っている」「足跡を残す」ということで，「みこたんを乗せたふぇにくすが走り回っている様子」にすることにしました．みこたんはいたずらっ子という設定なので，楽しそうに笑っていればなおよいでしょう．そういったシーンを頭に浮かべながら，メモに少し下書きをしてみます．

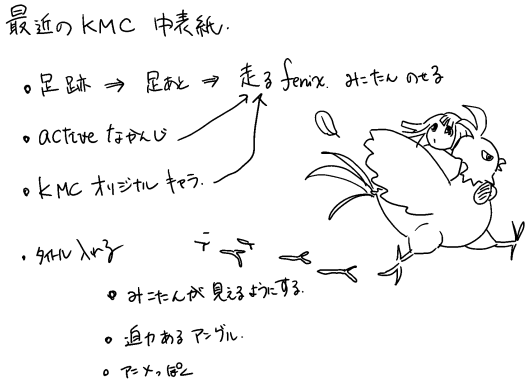


図 2　お絵かきメモ

この時点で，どんな塗り方にするか，どんなタッチにするかも考えておきます．今回は

コミカルな絵柄ですから，アニメや漫画のようににきっちりと線を描いて，塗りもそれに合わせてアニメ塗りにします．ぼかしやグラデーションはなるべく避けることにしました．

■カメラワークを決める　メモの下書きはそのまま使いません．思いつきで描いたアングルは極めてありきたりになりがちです．まず 3 つの候補を考えてみました．今回は中表紙

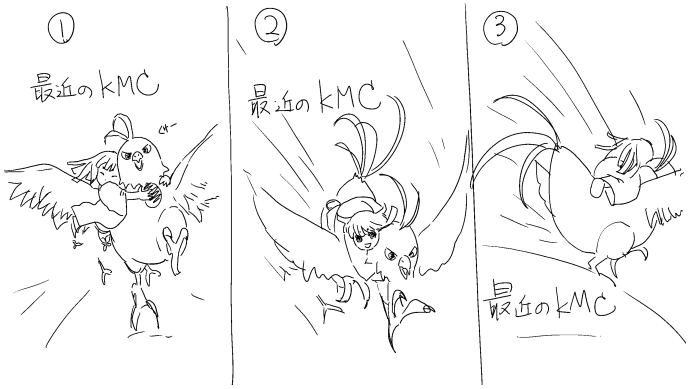


図 3　カメラワークの候補 3 つ

ですから，目立ってインパクトのあるものが求められます．イラストのテーマに沿うアングル，描き慣れないアングルを探す必要があるときは，先程のお絵描きメモを見ながら頭の中で場面をくるくる回して考えます[*9]．

ひねり出した 3 つの候補を設定したテーマに基づいて比較した結果，1 と 3 は「足跡が描けない」，1 は迫力があるが走っているというより飛んでいる感じになる，また突進のイメージになってしまう．3 は走り抜ける感じがあるが，そもそもキャラクターの顔が見えない．……という理由で却下になり，2 を採用することにしました．

■資料集めをする　ふぇにくすは大きな鳥[*10]なので，ダチョウの走っている画像などをあつめてきて羽や脚を描く時の参考にします．ダチョウっぽいキャラクターなども探して，これもひと通り見てみます．資料を参考にして，ふぇにくすの走っているシーンを練習しました（図 4）．

■下書きをする　ここまで来たら下書きをします．私はこれは紙でやることもあるし，コンピュータ上でやることもあります．今回はコンピュータでやることにしました．採用した構図をもとに，特に何も考えず描きます．走っているので，背景は野外で，草原かふつうの道にします．また羽を 1 つ 2 つ散らしてスピード感を演出します（図 5 左）．

[*9]難しいと思われるかもしれませんが，これもよい練習です．それにメモがあるととっかかりになるので，何もないところから考えるより易しくなります．

[*10]本当は考案された時は金色のニワトリだったのですが，私が気に入って描いているうちに巨大化してしまったのです．

図 4　資料をみて練習．左上・中央がダチョウの写真を見て写したもの

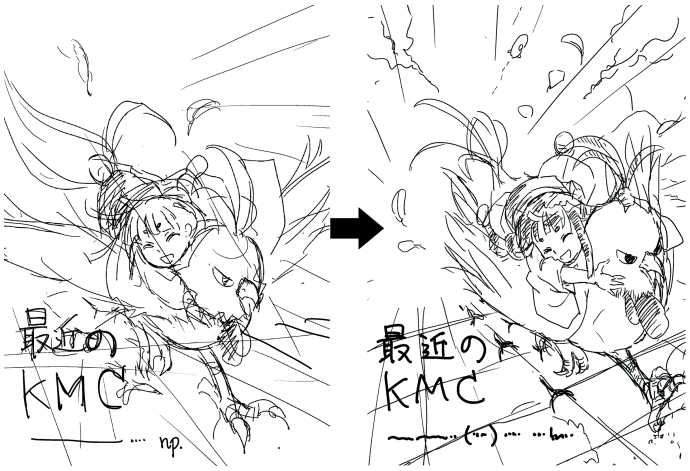


図 5　最初の下書き（左）と修正後の下書き（右）

■寝かせる　下書きのあと，ペン入れのあとには必須の行程です．描いた下書きに意識を向けない時間を作ります．1 時間でも，一晩でも，1 週間でもいいですが，慣れると短くできます．ココアでも一杯，キメましょう．

■修正する　もう一度下書きに目を向けると，客観的な視点が生じ，粗がみえてくるはずです．ここで細かいカメラワークの修正と，キャラクターの大きさの変更を行いました．ふぇにくすを斜めから見たほうがより迫力が出ますし（アンバランスさによる緊張），画面からはみ出んばかりの位置に置くと緊張感が生まれ，良くなります（走っている感じが出る）．また，空には雲を流して，よりスピード感が出るようにします（図 5 右）．

■線を描く・色を塗る　修正した下書きをしっかり点検して，これで変更点が無いことを確認したら，一気に線を描いて塗ります（図 6 左）[*11]．最初に決めた方針に従って何も考

[*11] 描き方や塗り方は今回の記事の焦点ではありませんので，割愛します．

えず進めていきます（何も考えなくていいところまで詰めるのがポイント）.

■仕上げ　もう一度この段階で少し寝かせてから，あらためて見直します．細かい修正（雲の描きこみ，砂埃の描き足し，陰影のエッジを立たせる）をし，集中線を入れてさらにスピード感・疾走感を強調しました．また走っているのにみこたんの髪がなびき具合が足りなかったので，もう少しおでこを出しました．これで完成です（図 6 右）.

■仕上げその後　最後にページ数を入れたり，作者のサインを入れます．サインを入れるといかにも「私の作品です」という感じになるので，ぜひ入れましょう[*12].

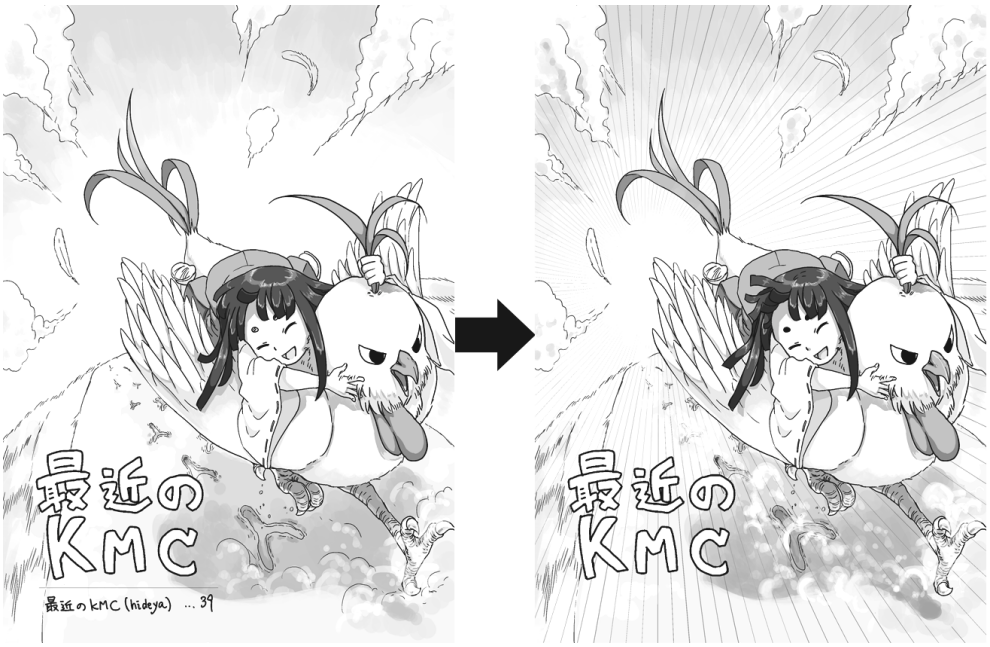


図 6　清書して色を塗ったもの（左）と，仕上げたもの（右）

なぜ設計が大事なの？

イラストには 2 種類あって，「何かを表現するために使われるイラスト」と，「ただ描きたいだけの・なんとなく描いてみたイラスト」があります．ゲームプロジェクトのグラフィック，宣伝のためのビラ，本の挿絵．こういうものに使われるイラストは全て前者であり、つまり道具です．道具は用途をはっきりさせておかなければただの役立たずになりますし，無駄な機能があればそれは邪魔なものでしかありません.

道具は開発段階において，試作→ユーザのフィードバック→試作→ユーザのフィードバック……と試行錯誤を繰り返して，完成します．道具としてのイラストも，これと同じ

[*12]完成品は「最近の KMC」中表紙をご覧ください．また今回は「あとがき」の中表紙も担当しました.

道を辿らなければ完成には至りません．そこでまず試作を行い，最初にユーザに見てもらいます．その時に必要なものは何か？それは，用途＝テーマです．

用途＝テーマに基づいた改善を行う

ひとりで楽しく描くだけなら，テーマなんて要りません．しかし人に見てもらう場合は「指摘」「批判」をもらう必要があります．その時にテーマが無ければ，「なぜこの物体を描いたのか」「なぜこの色を使ったのか」「なぜこんなフィルタを使ったのか」，こういった指摘ができません．テーマがあって初めて，そのテーマに沿っていない点を指摘できるからです．

先程の実際にお見せした行程でも，下書きの修正は当初のテーマである「アクティブな感じ」「疾走感」をより強めるための修正でした．そして，ひととおり彩色まで済ませたあとの修正も，「アクティブな感じ」「疾走感」を強めるために集中線を入れました．こういった細かい変更でさえ，テーマに基づいて行えばその理由の説明が可能になります．

イラストは感覚で描くものと思われがちですが，道具としてのイラストはある程度理詰めで作ることができます．理詰めで作るということは門外漢……つまり，イラストを見る側の一般の人とも議論が行えるということになり，より用途に合ったイラストを描くことができます．

迷いを捨てられる

テーマはイラストを描く側のためのものでもあります．ある物体を絵の中に入れるか迷っていたり，ある効果を出そうと思ってフィルタを使ったら，どちらがいいのかで悩んでしまったりすることがあるかと思います．そんな時，無駄に迷わずにテーマを振り返れば，よりそのテーマに沿ったほうを取る，ということができます．迷いを捨てて効率よく進むことができるのです．

意見をふるいにかけられる

始めから，「テーマに関する議論しかしない」と決めていれば，例えば「この画風は好みじゃない」とか「こういう表現は好きじゃない」という意見は退けることができます．「この画風はテーマに合わない」なら受け入れるべきです．主観を押し付けてくる意見と，イラストがテーマと食い違っていますよ，という意見は異なります．ゲームプロジェクトなどに必要なイラストを人目に晒すと，色々な意見が出ると思います．そこでテーマについて言及しているかでふるい分けをすれば，考える数を少なくできます．

さらなる上達のために

「明確に描く力」に加えて，「それっぽく描く力」を上げるための話をします．

模写をする

いろんな物を描きたいのなら，いろんな物を模写しましょう．「資料を集める」とほぼ同義ですが，普段から描き慣れないものを意識的に見つけて模写するクセをつけるとよいでしょう．人体構造なら Posemaniacs[*13]でいろんなポーズを見たり，隣にいるあなたの友人を使いましょう．また近場の風景の写真を撮ってきて，模写してそこに人物を置く，などのアレンジをした習作を描いてみてもよいでしょう．

■深入りは禁物　模写は何も考えずにすると時間をドブに捨てる行為となります．模写をする時には，「目的を設定する」「時間をかけたことを誇らない」点を心がけてください．うつし絵なのですから，時間をかければかけるほどより似せることが出来るのは当然です．しかしそれは絵の上達ではなく，単に細かい瑣末な部分のうつし絵がうまくなっているだけです．

建物の写真を模写するなら，窓の細かい造形などは省いて，どんな風にパースを取ればよいか，どんな風に描けばそれっぽくなるか．人物を模写するなら，人体構造はどうなっているか？など，目的を絞ってやるようにしましょう．細かいところに目が行ってしまうと，全体を見る力が無くなってしまいます．むしろ短時間で本質をぎゅっと掴んだような模写ができるようになるほうがよいのです．

■インスピレーションは過去の引き出しから出来ている　模写をしてたくさん引き出しが増えてくると，計画なしに白紙に向かってもさらさらっと何か描けてしまったりします．これがいわゆる「インスピレーション」による類のイラストです．冒頭で言及しましたが，プロのつくる素晴らしいインスピレーションによる作品は，ウン百回ウン千回，いやウン万回溜め込んだ引き出しの中の部品で出来ています．

鶏肉と野菜をうんと入れた水炊きの出汁のようなもので，具が多ければ多いほど味のある何かができてきます．ああ，鍋を囲みたくなってきた……．何の話だったでしょうか，そうそう，描き始めで全然具がなければ味気ない出汁にしかなりませんよね．とにかく，インスピレーションに夢を見てはいけません．鍋食いてぇ……．

QOL を上げる

「4 つのルール」のあたりで軽く触れた，線画を綺麗にしろ，という話をします．イラストでは塗りや色とともに，輪郭，つまり線が多くを語ります．

線の集合体ともいえる漫画を例に取りましょう．ほとんどがモノクロで，白い部分がとても多いにも関わらず，深く微妙なニュアンスを語る絵が漫画にはたくさんあります．そのニュアンスはコンテクスト・台詞などに拠る要素ももちろんあるのですが，線にも大きく拠っています．簡潔で，対象の本質をよく表した線にはいっそ芸術的なものすら感じられます．反対に，ごちゃごちゃと描き込まれたラフには脳を錯覚させるためのごまかしし

[*13] http://www.posemaniacs.com/blog/

か感じられません．脳の自己欺瞞にまかせず，自分の手で 1 本の線が引けるように練習しましょう．また，短い線を継ぎ接ぎで描いていくのではなく，1 ストロークをできるだけ長くできるようにしましょう．

個人的な経験では，線画を構成する線分の 1 本 1 本の長さと、「それっぽく描く力」のあいだには相関があるように思います．つまり長いストロークでさらさらと描けるほど上手，ということです．線を 1 本しか引けない・継ぎ接ぎもできないとなると，対象の本質は何か？対象を最も良くあらわすものは何か？と観察せざるを得なくなるからです．その結果，よい線画が描けるようになっていくのです．私はこれを「QOL (Quality of Line) を上げる」と表現しています．

少ない線は無駄な情報が削ぎ落とされているので，簡潔で美しく，本質を伝えるパワーがあります．すぐれたデザインの道具はたいてい無駄が削ぎ落とされて美しいものですが，線画でも同じことが言えるでしょう．

公開する

ひとりで描くより，人に見てもらうほうが意見を貰いやすく，上達も早くなります．ここでもテーマをしっかり押さえて，もらった意見をふるい分けしましょう．耳に痛い意見もたくさんあるはずですが，ぐっとこらえて改善目標にしましょう．

公開方法で主流なのは pixiv かと思いますが，手でアップロードしたりコメントをつけるのが面倒なので，私は Tumblr を利用しています．特定のディレクトリに PNG ファイルを保存すると，cron で回されているスクリプトが勝手にそれをアップロードするので，描けば描くほど勝手にイラストが増えていくという，（ある種恐ろしい）便利な環境にしています．

■宣伝　イラストを公開したら，Twitter・Facebook などの SNS でイラストへのリンクを張るのがよいでしょう．宣伝をする時は，イラストに関連すること以外の余計な情報を入らないように心がけましょう．他の粗雑な情報が入っているとクリックする気が失せるものです．

仕事をもらう

ゲームプロジェクトのグラフィックでも，Twitter のアイコンでもなんでもよいのですが，人に頼まれてイラストを描くととてもよい勉強になります．この場合は「相手の要求＝テーマ」ですから，むしろやりやすいかもしれません．人の好みに合わせてものを作るのは大変ですが，きっと実力が上がることと思います．

あとがき

図 7　Moko 近影

大学に入るまで何も考えずに好きなものだけを描く日々でしたが，KMC に入部してからは，「使うためのイラスト」を求められるようになり，今回お話ししたようなことを無意識に身につけていきました．昨年の冬から少しずつノウハウの言語化が進み，前回発行した独習 KMC Vol. β では人体の骨格についての話を書くことができました．

また今年度は「イラスト品評会」に TA 的な役割で参加し，毎週の課題をこなしつつ参加者に蜜柑を投げつけたりツッコミを入れるうちに，さらに多くのノウハウが見えて来，今回第 2 刊目の部誌ではこのような抽象的な話を書くことができるに至りました．

謝辞

今回の成果は，昨年冬に大変な成長曲線を見せた koji 君，短絡と諦めの堆積を繰り返しつつも上達した nojima さん，DTMer なのに何故か数々の挑戦的課題と厳しいツッコミをくれた tokiwa さん，「イラスト品評会」という場を作ってくれた Crys 君，意欲あふれる参加者の gire 君・madaragi 君・relsa 君・TJ 君，そして，私にいろんな仕事をくださったみなさんのおかげです．ありがとうございました．

■Twitter　moko_oxygen

■Web　O2-oxygen：http://moon.kmc.gr.jp/~moko/

■Tumblr　独習美術解剖学：http://mokograffiti.tumblr.com/

作曲において抑えるべき概念

tokiwa

はじめに

こんにちは．この項では，DTM（デスクトップミュージック：コンピュータを用いた作曲）について解説させて頂きます．さて，DTM において一番大事なことはなんでしょうか？作曲ソフトの使い方？環境？いいえ違います．作曲に関する知識・感覚です．漫然と絵を書いているだけでは CG が上達しないのと同様に，作曲においても何も知らず何も考えずに漫然とやってるだけでは DTM は上達しません．いきなり話がずれてしまっているようですが，これだけはどうしても外せない，大事なものなのです．

私は KMC の OB ですが，在籍期間の 6 年にわたりゲーム音楽の作曲を行うとともに，後輩の指導も行ってきました．それらの経験から考察した「作曲の基礎」についてのお話をさせて頂きたいと思います[*1]．

尚この記事は，最低限の音楽用語を理解していることを想定して書かれています．音楽知識がない方は置いてきぼりになってしまいますが，作曲を志している方を対象とした記事ですので，ご了承をお願いします．

音楽の構成要件

作曲をするためにも，作曲に関して議論を深めるためにも，まず必要なことは，音楽について理解を深めることです．とはいえ，音楽に関する基本的な話から初めてしまうときりがありませんので，ここでは作曲において踏まえておきたい音楽の構成要件について，簡単に述べさせていただきたいと思います．

さて，音楽の構成，と書くと，DTM ソフトの画面を思い浮かべる人が多いでしょう．縦軸にピアノロールないしトラック，横軸に小節，という形に展開するものが一般的でしょうか．今回ここで紹介する構成もまた，それに準じた形のものになります．図 1 に示します．

[*1] `http://moon.kmc.gr.jp/~tokiwa/` にて過去の作品を公開しています．

			時間軸に沿った展開							
			主題				主題			
			動機		動機		動機		動機	
			小節	小節	小節	小節	小節	小節	小節	小節
音階に沿った展開	高音 (C4-G5)	旋律 (メロディー)								
	最低音 以上(G3-)	和声 (ハーモニー)								
	最低音 (G1-G3)	ベース								
	無音階	リズム								

図 1　音楽の構成

音楽は，簡単には，時間軸に沿った展開と，音階に沿った展開の二次元情報となっていると考えるとよいでしょう．まさしくピアノロールそのままの構図となっています．

時間軸に沿った展開

まず，時間軸に沿った展開について述べていきたいと思います．「小節」は音楽を知っている人ならば概ねご存知でしょう，時間軸の最低構成となる単位です．楽譜の冒頭で指定している「4 分の 4 拍子」などというのは，この小節内に 4 分音符[*2]が 4 つ入りますよ，という意味となります．

さて，小節は時間軸の最小単位と言いましたが，つまり小節よりも大きな単位もあります．小節の次に大きい単位は「動機」で，おおむね 2 小節からなります．さらに大きい単位は「主題」で，2–4 動機からなります．「主題」レベルになると，曲の一部としての体裁がかなり整ってきます．例えば，J-POP の曲の部分を表す用語として，「イントロ」「A メロ」「サビ」などという表現が用いられるのはご存知でしょうか．主題はこのレベルのサイズだと思ってください[*3]．

主題は非常に重要な概念となります．なぜなら，このレベルになってくるとまとまりがよいため，ある曲の主題部分をまるまる別の曲に移植するなどということが可能になるためです．例えば曲を書いていると，この主題はいいけどこの主題はボツだな，などということがよく生じてきます．そういうときに，昔ボツにした主題を放り込んで軽く手直しをしたら見事な曲が出来たなんてことは，結構あることなのです．ここで大事なことは，曲を書いているとどうしてもボツネタが出てきますが，ボツだからといって決してデータを消してはいけません．ある日ボツにした主題が，数年後に自分の傑作の一部になっていた，なんてことは，十分にありえる話なのです．

[*2] 音符の分類の一つ．尚，MIDI のテンポの数字（おおむね 90–180 あたりで指定）は一分間あたりの 4 分音符の数を指定しています．

[*3] 厳密には，この単位になると 2 主題程度で構成される場合も多いですが．

また主題は区切りとしてキリがいいため，作曲作業や耳コピ（後述）の作業の一段落としてはいい単位になります．つい思いついたからといってつらつらと主旋律を書いても，後で他のパートを書いていくうちに，当初考えた主旋律が合わなくなってきて困ったり，とりあえず書いた主旋律に合う他パートが思いつかなくなったりということはよくあります．作曲作業は主題単位程度で進めるのがよいでしょう．途中で詰まっても，全く別の主題を新しく作ってそれと組み合わせるという解決もありえるのです．

音階に沿った展開

次に，音階に沿った展開について述べていきたいと思います．作曲を行う上で欠かせないパートというものがいくつかあり，私はこれを「主旋律」「和声」「ベース」「リズム」と分類しています[*4][*5]．この 4 パートが音楽の最小構成だと押さえておくことが大事です．この 4 パートさえ押さえていれば，各パートで鳴っている音が一音ずつでも十分音楽としての体裁を整えることが出来ます．例えば，任天堂のファミリーコンピューターの音源部は，同時発音数が 5 つしかなく，その楽器編成も矩形波，矩形波，三角波，疑似ノイズ，DPCM[*6]が一音ずつという極めてシビアな制限を受けています．しかし，このファミコン音源を用いて作成された曲でも，上記の最小構成を守ることにより，十分に音楽としての体裁を保つだけでなく，高い評価を受けている曲が多数あります．

パートについて，分かりやすいものから解説していきましょう．まずは「主旋律」（メロディー）についてです．主旋律は音楽において最も「耳につく」部位であり，音楽には欠かせないものです．主旋律は，おおむね中央のド (C4，周波数 242Hz) から，1 オクターブ上のソ (G5) までの間に収めるとよいでしょう[*7]．主旋律は最もよく耳につく部分であるだけに，主旋律の音の変化があまりないとつまらなくなってしまいます[*8]．また主旋律は，よく動く部位だけに，どの音を使ってよいかをちゃんと意識しないと，不協和音を鳴らしてしまう，ということがよく起こるため，注意が必要です．

次に「ベース」についてです．ベースは最低音域をカバーするパートになりますが，これも適切な音の高さがあります．基本的には普通のソの 3 オクターブ下のソ (G1) と 1 オクターブ下のソ (G3) の間あたりがよいでしょう．

そして，「リズム」についてです．リズムとはリズムを刻む為のパートで，打楽器などを主に使用します．特徴としては，基本的には音階がないことです[*9]．リズムパートにおいては，ドラムの音が汎用されます．そのため，ドラムを用いたリズムの刻み方を知って

[*4]私の認識に既存の用語を当てはめているので，各用語の正確な意味とはニュアンスがやや異なります．特に和声は，ここでは対旋律＋和音くらいのニュアンスです．

[*5]尚，旋律・和声・リズムの 3 つを，西洋音楽では音楽の三要素と呼びます．

[*6]差分パルス符号変調。後述する PCM 音源と同様、サンプリングした音を再生するものです．

[*7]ここの C4, G5 は，国際式オクターブ表記です．C, G のアルファベットが音名，4, 5 の数字がオクターブの高さです．

[*8]主旋律をまるまる 1 小節同じ音階の音で刻むだけの曲を書く癖がある人を見たことがあります．ドードードーード　ドーレーレーーレといった感じに．

[*9]打楽器は言うに及びません．ファミコンでは疑似ノイズを打楽器がわりに用いることがあります．

おくと，リズムを書きやすくなります．一般的には，低音系のリズムとして，バスドラムとスネア，高音系のリズムとして，ハイハットとシンバルが汎用されます．

最後に，「和声」（ハーモニー）についてです．和声はここまでに書いたものに入らないパートだとお考えください．すなわち音域はベースより上の全域となります．また，曲をより賑やかにしたい，より厚みを増したい場合は，この和声パートを増やすことで対応します．

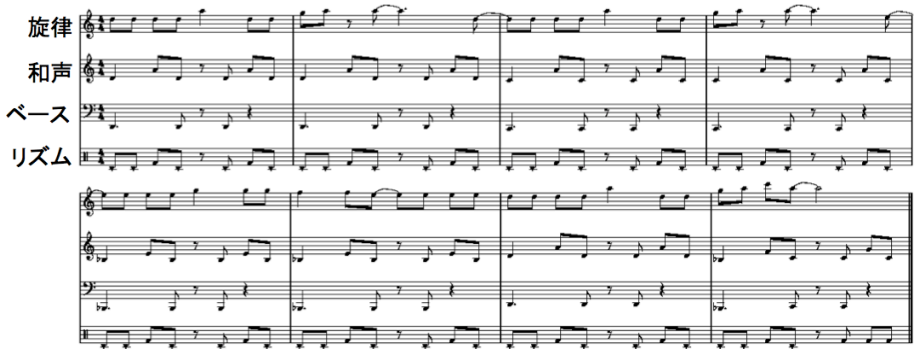


図 2　最小構成の楽譜の例．拙作「流浪の料理人」冒頭の主題．

和音について知るべきこと

さて，曲を書く上で意識しなければいけないことがあります．それは和音（コード）です．和音とは，音楽をしている方ならご存知でしょうが，複数の音名の音が同時に響く音のことです．

作曲する上で和音について知っておくべき大事な点は 3 つあります．まず，「和音は限られたパターンしか存在しない」こと．次に，「同じ和音でも，転回・分散により様々な表現が可能」なこと．そして，「小節ないし半小節は和音の支配を受ける」ことです．

和音は限られたパターンしか存在しない

まず一つめについて．ご存知の方もいらっしゃるでしょうが，和音には名前がついています．和音は，ある音（根音）の音名と，その根音を元として 3 度・5 度・(7 度・9 度）上にどういう音が鳴るかという組み合わせにより名前が決まります[*10]．

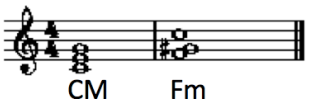


図 3　和音には一意に名前がつく

例を挙げましょう．ドを根音として，長 3 度上のミ，完

[*10]根音と 3 度・5 度上の音が同時に鳴る和音を三和音と呼びます．三和音に根音の 7 度上の音が重なると四和音，四和音に根音の 9 度上の音が重なると五和音です．

全 5 度上のソが同時になると C メジャー (CM)，ファを
根音として，短 3 度上のラ♭，完全 5 度上のドが同時になると F マイナー (Fm) となります[*11].

ここで大事なことは，名前があるということはすなわち，和音というのは鳴らせるパターンが決まっているということです．つまり，好き勝手に音を鳴らして「和音だ！」と言い張ることはできないのです．逆に言えば，既存の和音のパターンに沿って音を鳴らせば，美しく音を響かせることができるということです．

同じ和音でも様々な表現ができる

次に 2 つめ，同じ和音でも様々な表現が可能ということについてです．右の楽譜に 4 分音符で表された 3 つの和音と，16 音符で表された早弾きが 1 つありますが，これは全て同じ C メジャーの和音です．

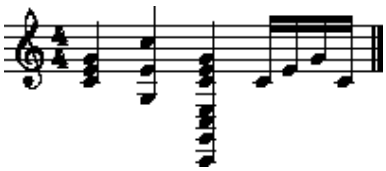


図 4　全て同じ C メジャー（ドミソ）の和音！

和音とは，複数の「音名の」音が同時に響くと先に書きました．これはすなわちどういうことかというと，音名が同じなら別のオクターブの音が鳴っていようが，複数のオクターブに渡って同じ音名の音が鳴っていようが，同じ和音だということです．尚，このように同じ和音でも多様な鳴らし方が出来るということは，和音の最低音に和音の根音ではない音が来る場合があるということでもあります．これを和音の転回と呼びます．和音が転回すると多少和音の性格が変わるので，同じ和音を鳴らしている場合でも，転回を行うと雰囲気を変えることが出来ます．

最後の 16 音符で表されたものは，分散和音（アルペジオ）と呼ばれるものです．和音の各音を同時に鳴らさず，このように順次鳴らすことでも，音の残像効果により和音として機能させることが出来ます．

和音は一小節ないし半小節を支配する

そして 3 つめ，和音は一小節ないし半小節を支配することについてです．和音は，おおむね一小節ないし半小節単位で鳴ります．その間，上に述べた主旋律，和声，ベースの音は，基本的に和音に帰属する音にならなければなりません．

無論このままでは，音楽と言っても和音をジャンジャン鳴らすだけのつまらないものになってしまいます．そのため，ちょっとくらいなら和音から離れた音（非和声音）を鳴らすことは差し支えありません．但し，それにもパターンがあることを覚えておいてください．簡単には，各小節の最初または最後の音（あるいは両方）は基本的に和音に帰属させ

[*11]尚，ここで記載している和音名は「コードネーム」と呼ばれる表記法です．和音の表記には他にも「和音記号」と呼ばれる，ある調の中での役割を考えた表記法もあります．

る，くらいの心構えでいればいいでしょう．また，ちょっとくらいなら和音から離れてもよいと書きましたが，複数の音が同時に和音から離れると，和音がめちゃくちゃになってしまいます．和音から離れていいのは主旋律や和声の一部くらいに思っておくとよいでしょう．

尚，この一小節ないし半小節単位での和音をどう繋いでいくかのパターンを，コード進行と呼びます．ここでは詳述しませんが，これに関する知識があると，作曲が大いに楽になるでしょう[*12]．

和音についてより詳しく知りたい方は，Wikipedia の和音の記事が秀逸で参考になります．和音の種類や非和声音についての細かな分類・サンプルなどもあるので，確認されるとよいでしょう．

初心者が陥りがちな罠

初心者にありがちな悩みとして，自分の思いついた曲をとりあえず書いてみたものの，とても聞くに耐えないものができてしまったり，どうも聞き心地が悪く変なところがあるものの，どう修正すればよいかが分からない．というものがあります．無論ある程度は経験を積まないと対処できないものです．

しかし，その中でも「初心者が犯しがちな致命的なミス」があります．私はこれを「禁忌のリスト」として，指導においては必ず指摘対象としてきました．正直なところ，これを改善するだけでも，「人様に聞かせられる曲」レベルには非常に到達しやすくなります．

禁忌のリストの内容は 3 つあります．「不協和音を残す」「ベースを書かない」「和音を無視する」です．

不協和音を残す

まず「不協和音を残す」，これははっきり言います，論外です．理由は簡単，普通の人でも聞いていて変だと気づく上に，聞く側が不愉快になるからです．

不協和音で初心者がやりがちなのは，別パートの音（例えば主旋律と和声）を半音ないし全音被らせる，というのがあります．聞いていて違和感を感じたら，自分の曲の楽譜をしっかりと確認しましょう．修正する場合は，違和感のある小節ないし半小節で鳴っている和音はなにかを確認し，それからズレている音を探して，その和音の構成音に変更すると，違和感を解消しやすいです．どうしてもその小節ないし半小節の和音が分からない場合は，最後の手段としてその音を消してしまいましょう，曲の厚みは少なくなくなりますが，違和感は解消されます[*13]．

[*12] 作曲が初めてだという人が，コード進行の講義を受けてすぐさまモノにし，初めての作曲とは思えないハイレベルな音楽を書かれていたのを見たことがあります．

[*13] 四和音ないし五和音は，基本となる三和音ないし四和音を抜くと，構成音が不協和音になりやすいので，慣れないうちは避けたほうがよいです．また，プロフェッショナルの楽曲で不協和音を敢えて残していることがありますが，あれは基本を抑えているから許される高度な技術です．素人がやってはいけません．

ベースを書かない

次によく見られるのは，「ベースを書かない」です．ベースは音楽を聞くときにはあまり意識しないパートですが，曲を作る上では欠かせない「縁の下の力持ち」です．手持ちの音楽を音楽再生ソフトのイコライザーを使って低音部を完全に消す，あるいは MIDI ファイルならベースパートをミュートして再生してみてください．恐らく何か気の抜けた，聞くに耐えない音が聞こえると思います．

何故初心者がベースを書かないのかというと，聞いていて目立たないパートなので，必要を感じないためです．ベースの音が思いつかない場合は，各々の小節の和音の構成音を主旋律の 2 オクターブ程下に置いてみてください．それだけでも劇的に変わるでしょう．

和音を無視する

3 つ目が「和音を無視する」です．ベースも書いてある，目立つ不協和音も見当たらない，しかしなんか変だ，という場合によく見られます．例としては，和音構成音と主旋律が合っていない，というものが挙げられます．例としては，「ドミソ」の和音が鳴っているところに「レドレシ」と主旋律を鳴らす，などでしょうか[*14]．和音の項で述べた，「各小節の最初または最後の音（あるいは両方）は基本的に和音に帰属させる」を思い出しましょう．

耳コピ——作曲力を鍛えるための訓練

作曲をやっていると，「あっ，いいメロディ思いついた！」と思い，勇んで書いてみると，「あれ？俺が書きたいのはこんな曲だったっけ……」「なんか書いているうちに何が書きたいのか分からなくなってきた……」などということはよくあります．他にも，ベースやドラムなど，普段あまり意識していないパートの書き方が分からない……などということもよくあります．

そんな貴方におすすめなのが耳コピです．耳コピは作曲を志す人なら一度は聞いたことがある言葉だと思いますが，要するに既存の音楽を耳で聞き取って，自分で楽譜を再現する作業です．耳コピを繰り返すと，耳にした音や頭に思い浮かんだフレーズを正確に楽譜にアウトプットすることが出来るようになり，作曲のスピードと正確性が向上します．また，耳コピは既存曲を自分の手でなぞる行為になるので，作曲する上でプロがどういう構成で音楽を書いているのかを直にたどることが出来ます．このような点から，耳コピは作曲をする上で非常にためになる訓練になります．

[*14] ある程度，前述の不協和音と被るところがありますが．

chiptune のススメ

さて，耳コピというと，ついつい自分の好きな曲を耳コピしよう！と思うものです．しかしこれが大きな落とし穴となります．好きな曲なら耳コピのモチベーションは持続しやすいのは確かなのですが，何も考えずに趣味で曲を選んでしまうと，思った以上に複雑で，音が分からない，聞き取れないということが発生します．その結果，すぐに挫折したり，自分の聞き取れる部分だけ（例えば主旋律だけ）聞き取って満足したり，全然聞き取れていないのにとりあえず楽譜を置いてやり終えたような気分になってしまいます．これでは訓練になりません[*15]．

そこでお勧めするのが，chiptune の耳コピです．chiptune とは，1980 年代から 90 年代前半の頃，まだゲームハードのマシンスペックが貧弱で音数に制限が厳しかった頃のゲーム音楽を指します[*16]．例としてファミリーコンピューターを挙げると，サウンドの制約は音色 4（矩形波，三角波，疑似ノイズ，DPCM），同時発音数 5 となっています．しかもこの範囲でゲーム中の効果音も賄わなければならず，非常に厳しい制約が課されていることがおわかりでしょう．

chiptune が教育上優れているところは 3 つあります．まず，制限が厳しいので，耳コピが簡単なことです．耳コピの入門用としてはうってつけです．自分のコピーしている曲がどのような制限のもとで書かれているかを知っていれば，とりあえずコピーしてみた物と原曲を比べて，自分の曲に何が足りないかを類推しやすくもなります[*17]．次に，こういった最小構成とも呼べるレベルの制約の中でも，曲としての体裁を保っているので，曲を作る上で重要な構成というものを理解することが出来ます．そして，このような厳しい制約の中でも熱烈なファンを持つほどの優れた音楽を生み出しているという実例を示していることです．

chiptune の中には，フリーソフトで同様の音源が実装可能なものもあり[*18]，理論的には原曲と同等の音源環境で作曲することも可能です．但し注意としては，chiptune の曲の音色は，味を出すために細かな調整を施されている場合が多く，音色を正確に再現することは困難です．訓練として耳コピを行う場合，重要なことは音色を合わせ完全に音を再現することではなく，正確に音階を取ることなので，そこを履き違えないようにしましょう[*19]．

chiptune レベルでは流石に物足りないという自信のある方には，正確な chiptune の定

[*15]耳コピだと言って持ってきたものを聞いたら，音階が目茶苦茶だった，なんてことはザラにあります．

[*16]正確には，それらを模して作られた音楽も含む．

[*17]同時発音数が 5 だと知っているのと，何音同時発音しているか分からない状態では，前者のほうが自分のコピーの完成度を推し量り易い，ということです．

[*18]例：ファミコンと同様の音楽を作成可能な Famitracker，フリーソフト「洞窟物語」の音楽エンジンで作曲可能なオルガーニャ．

[*19]chiptune の耳コピで，音色探しに拘ってみたものの，肝心の音取りが残念だった人を見たことがあります．

義からは外れますが，スーパーファミコンのゲーム音楽などもお勧めです．こちらも音源こそ PCM 音源[*20]ながら同時発音数 8 と，かなり制約が厳しい傾向にあります．さらにそのうち 2 音はドラムに使っていたり，そもそも 8 音全て使っていないなどということもザラにあり，比較的コピーが簡単なのです．

耳コピにおいて大事なこと

さて，耳コピにおいて大事なことは 5 つあります．それは「執念」「度胸」「知恵」「相対音感」「再生環境」です。

■執念　執念は耳コピにおいて最も重要なものです．耳コピでは，簡単に聞き取れない音を何度も何度も何度も何度も繰り返し聞いて音を取っていく必要があります．どんなに聞き取れなくても，心が折れそうになっても，完成させてやるという不屈の執念がなければ耳コピは出来ません．心を強く持ちましょう．

■度胸　どんなに頑張っても聞き取れないことがあります．諦めてさじを投げたくなりますが，逆に考えましょう．そう簡単に聞き取れないということは，案外誤魔化しが効きます．わからないパートは，和音などから類推して適当に音を置くだけで案外誤魔化せます．しかし，誤魔化すと言っても，禁忌リストにあるような状態で放置してはいけません．誰が聞いても欠陥に気づきます．

■知恵（音楽知識・工夫）　聞けないものをなんとか聞くためにはちょっとした小手先のテクも必要となります．特定の音域のパートが聞き取れないというときは，イコライザーを使って聞きたい音域以外の音を消してしまいましょう．その音域が低音の場合は，再生速度を 2 倍にすると，音階が 1 オクターブ上がり，聞き取りやすくなります[*21]．16 音符が連続するような早弾きにおいては，再生速度を下げると聞き取りやすくなります．また，和音の知識も役に立ちます．早弾きの部位が聞き取れなくても，その部位の和音が分かれば，和音構成音から早弾きで鳴っている音が類推できます[*22]し，分からない場合も和音構成音を利用すれば誤魔化しが効かせやすくなります．

■主旋律が聞き取れる程度の「相対音感」　相対音感はほとんどの人が備えています．聞き取り能力はある程度は才能や音楽経験などに依存しますが，足りない分は訓練あるのみです．

■再生環境　音と音の高さの区別が付きにくいとか，特定の音域が全然聞こえないようなスピーカーでは耳コピは辛いものがあります．耳コピには，いわゆるモニター用スピーカーと呼ばれるような，原音再現に定評のあるスピーカーを選ぶと良いでしょう．尚筆者

[*20] サンプリング音源．予め録音した音を音源として，音階などを調節して鳴らすもの．
[*21] 人間，2 つの音の高さの違いは低音同士より高音同士の方が聞き取りやすいのです．
[*22] 早弾きは実際は分散和音で構成されている場合が多いため．

は，スピーカーですと Timedomain mini か Olasonic TW-S7[*23]，イヤホンですと Sony MDR-E888 をお勧めしています．スピーカーは置き場所も重要です．いいスピーカーも，何も考えずにカーペットに直置きして音が変になっている，なんてことがよくあります．音が変だなと思ったら，スピーカーの置き場所を変えてみたり，スピーカーの置き台になるようなもの[*24]を置いてみると，音が良い方に変わるかもしれません．

音楽をデザインする

さて，作曲をするときにやって欲しいことがもう一つあります．それは音楽をデザインすることです．音楽をデザインするとはどういうことでしょうか？簡単にいえば，「これから作る音楽のコンセプトを明確にする」ことです．これは，ゲーム音楽のような，どういう用途の音楽を作るかを確定している場合は比較的簡単になりますが，そうでない場合も積極的に行っていって欲しいと思います．

なぜ音楽のコンセプトを定めないといけないのでしょうか？その理由は大きく 2 つあります．

1 つ目は，作曲が簡単になるからです．何もないところからとりあえず音楽を生み出すよりも，目的を限ってしまったほうが，その目的を達成するために何が必要か，と具体的に考えることができるようになり，構成を考えやすくなります．

2 つ目は，人の意見を聞いたり，音楽の品質について議論するときに，話がしやすくなるからです．例えば，パッと曲を渡していい曲かどうかを聞いたり，改善点を聞いても，相手は「音楽そのものの質」としてしか議論ができません．しかし，ここに「コンセプト」や「用途」といった概念を加えると，より多くの話を引き出すことができるでしょう．こうして批評すべき観点を増やすことで，より様々な角度から音楽を議論することができます．

コンセプトを定める

さて，コンセプトを定めるとして，どのように定めていけばいいのでしょうか？一番最初にするべきことは，「目的を明らかにする」ことです．ここでは例として，「戦車が出てくる RPG の戦車搭乗時のフィールド BGM」を作ることを目的とする場合を考えましょう．

目的が決まったら，まず最初にすることは，「目的を充足するには何が必要かを考える」ことです．

目的を充足するために考えることは大きく 2 つあります．まずは表現するべきものは何かです．これはその曲が流れる状況，その曲がなさねばならない演出，キャラクターのテーマならそのキャラクターの性格などになります．

[*23] USB スピーカー．

[*24] インシュレーターとも言います．市販品もありますが，筆者はティッシュの空箱や本で代用しています．

もう一つが曲としての体裁の制約です．例えば曲長や，ループせねばならないか否か，などです．単独の音楽作品であれば曲長を数分と長めにとり，曲を完結させねばなりません．ゲームで使うジングルなら曲長は数秒でよくループはありません．ゲーム中のBGM ならば曲長は 30 秒から 2 分程度でループをさせるとよいでしょう．同じゲーム中の BGM でも，アクションのある部位の BGM と，文章を読んだり熟考をするシーンのBGM では，テンポや曲長も違ってきます[*25]．

今回の例ですと，戦車に乗って荒野を走りまわり敵を倒しまくるというシチュエーション，フィールド BGM なのでエンカウント間の繋ぎ程度の長さ（長くて 1 分程度）があればよく，ループが必要であろう，というところでしょう．

そして次に，「何を表現するかを考える」ことです．ここで表現すべきは五感です．シチュエーションを想定し，その時自分が抱いている感覚を表現しましょう．表現できる感覚としては，視聴覚・触覚・嗅覚・思考・感情があります．今回の例であれば，目の前に広がる荒野，身を打つ砂埃，戦車のエンジン音，戦車に乗ってれば雑魚など敵じゃないという自信，というところでしょうか．尚，ここでキャラクターのテーマを表現したい場合は，そのキャラクターの抱いている感覚か，そのキャラクターを見ていて感じる感覚を表現すればいいのです．

レイアウトを考える

さて，ここまで考えたら，次は曲のパーツをどのように配置するかというレイアウトを考えましょう．要するに，ここで出てきた「表現したいことリスト」をどうすれば適切に表現できるか，を考えるのです．

例えば，戦車のエンジン音を表現したい，となれば，ベースを 8 分音符で刻み，エンジンの「ドドドドドドド」という音を表現する，という具合です．荒野・自信を表現するには，ディストーションが強くかかったエレキギターを使い，和音はメジャーコードを使おう，パワフルさを演出するために，音が強めのドラムキットを使おう，といった具合です．ちなみに，こうして完成した曲は，「Dust Cloud」というタイトルで公開しています．興味のある方はご鑑賞ください．

レイアウトをする上で意識するべきことは，「このパートはこういう目的のために必要なパートだ！」と説明できるように意識しながら，パートを増やしていくことです．こうすることで，趣旨の曖昧なパートを削り表現すべきテーマを明瞭にできるだけでなく，パートを増やしすぎてゴタゴタしたときも躊躇なく不要パートを削る決断ができるようになるでしょう．

また，自分が普段あまり考えないような曲のレイアウトをしようとすると，やはり関連した音楽・似たようなテーマの音楽を参考にすることが必要になってきます．今回例に上

[*25] そんなの当たり前じゃん，と思う人もいるかもしれませんが．シューティングの一面（ステージ部 45 秒）に当てる想定を指定したら 2 分の曲を書いてきた人や，アイテム一つ取得すればゲームの状況が一気に危機的になるようなアクションゲームにローテンポ長調の BGM を持ってくる人がいました．

げた曲についても，私は普段暗めのオーケストラな曲を書いているので，実のところこういう曲目は苦手です．そこで今回は，参考として映画「TAXi」のメインテーマなどを聞いています．今回のレイアウトも，この曲を聞かなければ思いつかなかったところがあります．

おわりに

さて，いかがでしたでしょうか．読んでいて，「音楽って案外縛りだらけで自由じゃないな」と思った方もいらっしゃると思います．そう，はっきり言って音楽なんて縛りだらけです．縛りを外れるとすぐに聞くに耐えない不愉快なものが出来上がってしまいます．

しかし，この世界にはこんな縛りの中でも様々な素晴らしい音楽が満ち溢れています．人が法の下でも素晴らしい人生を送れるように，音楽も縛りの下でも素晴らしい奏で方が出来るのです．

それでも，こんな縛りだらけの中で，一歩を踏み出すのは怖い人もいるでしょう．ここで，作曲においてここまでに書いていない大事なことをもう一つ書きたいと思います．いろんな曲を，咀嚼するように聞きましょう．そしてそれを肥やしとして音楽を作りましょう．

この世に真のオリジナルなどないのです．この世の音楽も，盗作とまではいかないまでも，どこかで聞いたような旋律の組み合わせ構成されたような，既存の音楽のオリジナルブレンドとでも言うべきものがたくさんあります．それでも多くの人に愛されています．

あなたの血肉を構成するのはあなたの食べたものであるのと同様に，あなたの音楽を構成するのはあなたの聞いてきたものなのです．制約に恐怖することなく，既存の音楽から学び，あなた自身の作曲の道を切り開いていきましょう．


最近の
KMC
hideya
41
Moko

最近の KMC

hideya

始めに

この記事は，前回の部誌発行以降の 2011 年 1 月からの我々 KMC が過ごした時間を皆様にお伝えするものである．

年明け〜春合宿

大学生にとって年明けの時期とは，やたら創作意欲が湧く[*1]試験期間とそれを無事（かどうかは人による）切り抜けた者を祝福する長い長い春休みである．

KMC の春休みといえばコタツとマシンとみかん[*2]とボードゲーム……はさておいて，春合宿である．説明しておくと，春合宿とは例年春休みに行われる大きな合宿であり，各部員の深い知識を共有するための講座が主として行われ，それ以外にも新勧期に向けてのミーティングや OB も交えた酒うま[*3]，各部員の趣味をさらけ出す裏講座などが行われる．後半は割とどうでもいい．

今年の春合宿の思い出は，何と言っても大雪．山中の合宿施設に着くや否や，雪に足を取られる部員達．ロッジに荷物を下ろし次第，慣れない肉体労働＝雪かきをする部員達．そして低い気温と反比例する高いテンションで雪に埋まる部員達．

そんなこんなで，我々はメインとなる部員による講座を進めていった．今回特に人気があったのは，possum 氏による物語工学に関する講座．そのタイトルの長さに恥じない観客動員数を記録していた．他にも画像フォーマット，自然言語処理，マルクスと社会主義など様々なテーマの講座が行われた．

無事行方不明者もなく春合宿を終えた我々はとりあえず飲み会[*4]をし，来たるべき新勧の準備を本格的に始める事となる．

[*1] 現実逃避．

[*2] 諸事情により，KMC ではみかんは飲み物である．

[*3] 酒が美味い，の略．つまり飲酒．この酒うまの登場人物は全て 20 歳以上です．

[*4] 大学サークルの宿命．

図 1　雪は人に魔法をかける．

図 2　枠幅拡張（レイアウトブレーカー）

新勧準備期

新勧期には，新入部員を掻き集める努力をしなければならない．これは，大学サークルの定めである．

まず行ったのは立て看板の準備．絵柄自体は京都大学 11 月祭の時の物を流用しつつ，みこたん[*5]のセリフを差し替える形となった．作業に携わった部員は職人気質だったらしく，フォントを美しく看板に写し取る事に成功した．

加えて，存在感を引き出すために欠かせない手持ち看板を一つ増産した．新しい看板は耐水性の獲得に成功．我々の技術力は確実に上がってきている．

また，部室の壁紙が古臭くて陰気臭い，窓ガラスのヒビがエレガントでない，という意見が出た事もあり部室の壁紙張り替えと窓ガラスの交換を行った．窓ガラスの交換は業者

[*5]KMC のマスコット……なのかどうかは定かでは無い．

に依頼したが，壁紙の張り替えは全て部員の手で行われた．部室マシン群の異常な程のタコ足配線に溜息を吐きつつも，DIY 精神を発揮して真っ白な壁紙に張り替えてくれた部員を，改めて労いたい．

そして定番のビラである．今年は 6 種類ものビラを用意した．入試問題を模したビラ，アニメキャラの画像を使ったビラ，某ゲームの演出を意識したビラなどが用意され，見る者を楽しませた．

図 3　シューターの呼び水になったのかは不明

これらの多彩なビラは果たして新入部員を呼び込んだのだろうか……？それは神のみぞ知る．

新勧本番

4 月．様々なものがスタートする時期であり，大学サークルにとっては熾烈な戦い，そう新勧の幕開けである．我々 KMC もより多くの部員を獲得するために現役部員総動員でビラまきなどの新勧活動を行った．決してより多くの部費を吸い取る為ではない．部員が多い事の素晴らしさ（と，その逆）はよく知っているからだ．週 2 回の例会を新入生説明会とし，KMC を紹介するプレゼンを行い[*6]，晩御飯を奢り[*7]，部室を紹介する．文章にしてしまえば少々の事だが，現実のそれは確実に我々の体力・精神力を削っていった．

しかしこれらの苦行の結果，今年の新入部員は 30 名を上回った．大勝利である．

熱き志を持った仲間を増やし，我々は再び日々精進していくこととなる．

[*6]何回も何回もやるので，飽きる．

[*7]諸事情により，この時期は会長と会計は晩御飯が食べられない．辛い．

図 4　この写真は部屋の「中央」から撮影されたものである．

大学でいうと前期日程

新入部員達もだんだんと馴染んでいき，KMC メンバーはいつものように各自の目標に向かって自分を高めていった．この辺りの活動はブログ[*8]にも書いてあるので，詳しく知りたい方はそちらを参照して頂きたい．

そして今，我々は夏のコミックマーケットに出すための部誌の執筆に取り掛かっている．前回とは違う編集長と執筆環境[*9]の中，各自の得意分野に関する内容を書き下している．あなたがこの文章を読んでいるという事は，無事部誌が完成したという事だろう．我々の苦労の結晶を，どうか大事にして頂きたい．

さて，時系列が現在に追い付くどころか少し先まで行ってしまった．そろそろこの記事ともお別れである．

最後に一言．我々 KMC はこれからも成長し続けるであろう．日本や世界の未来を担う若者の成長を，これからも皆さんに気にして頂ければ幸いである．

それでは，またいずれ．

自分で出した勉強会の課題に苦しみつつ
KMC 第 34 代会長 hideya

[*8] http://d.hatena.ne.jp/kmc-log
[*9] 今回の記事は全て Tex で書かれています．

部員のコラム

KMC 部誌製作過程

seikichi

はじめに

KMC 4 回生の seikichi です．このコラムでは KMC の部誌製作過程を解説します．コミックマーケットや学園祭で会誌を頒布している部やサークルはたくさんありますが，どのように作られているかが話題になることは滅多にありません．このコラムが他の団体の参考になれば幸いです．

方針

KMC は毎年 3 月の末に京都の山奥で合宿を行っているのですが，今年はそこで部誌の編集長と方針を決定しました．前回の部誌は KMC というサークルを紹介する内容が中心でしたが，今回は技術的な内容を増やして外部の人が楽しめる部誌を目指そうということになりました．

詳細な日程は 5 月の末に次のように決めました．

5/23	原稿担当決定
7/4	原稿〆切
7/6–8	校正
7/9–10	製本

構成

前回の部誌は編集長が記事[*1]を集めて Pages[*2]で整形していました．ただ Pages を使える部員が数名しかいなかったため，修正を即座に反映することが難しく，また編集長に負

[*1] 大半がプレーンテキスト．

[*2] http://www.apple.com/jp/iwork/pages/

担が集中してしまう仕組みでした．今回はその問題を解決するため，LaTeX で部誌を製作することになりました．

校正

無事に記事が集まったところで校正が始まりました．今回は Redmine[*3]というプロジェクト管理ソフトウェアを用いました．誤字脱字などは見つけた人が修正し，執筆者の判断が必要な部分は Redmine に「バグ」として問題を登録します．校正の仕組み自体はこれで問題無かったのですが，今回の部誌は技術的に難しい内容が多かったためか，部員同士で記事をチェックするのが一苦労でした．

印刷

当初は部室で部誌を印刷し，人手で製本する予定だったのですが，出来上がった部誌は 100 ページを越える大作だったため，印刷会社に印刷と製本を急遽依頼することになりました．

反省

今回は部誌のバージョン管理に Git[*4]を利用しました．ただ部員が Git にまだ慣れていなかったためか，複雑なコミットログが生成されてしまいました．結果的に 20 人以上が関わるプロジェクトだったので，もう少しバージョン管理に気を使うべきだったと反省しています．なお KMC では少し前までバージョン管理と言えば SVN[*5]が主流だったので，そちらを利用した方が良かったかもしれません．

スケジュールは余裕を持って組んだつもりだったのですが，校正の期間が長引いたり，印刷を急遽変更したため，結局ギリギリの日程になってしまいました．次に部誌を作ることがあれば，編集長には今回の経験を活かしてもらいたいです．

感想

前回とやり方を変えたので，上手くいくか不安もありましたが，無事完成して良かったです．ちなみに編集長としての個人的な感想としては，バージョン管理システムを導入したおかげで「途中までででも良いから記事をコミットして下さい」と気軽に言えるので，記事を催促し易かったです．

[*3] http://redmine.jp/
[*4] http://git-scm.com/
[*5] http://subversion.apache.org/

Suffix Array を使った高速検索

nojima

はじめに

巨大な文書セットに対して全文検索をかけたくなったりとか，めちゃくちゃ長い DNA から特定のパターンを抜き出したりしたくなることは，きっとよくあると思います．この記事は，このような巨大な文字列から小さな部分文字列を高速に検索する方法と，その C++ による実装を紹介します．

文字列検索問題

次の問題を考えます．

> 長さ N の文字列 A と，長さ P の文字列 W が与えられる．A に対して前処理をすることで，A が W を部分文字列として含むか否かを高速に判定せよ．

この問題では，検索文字列 W が長い文字列 A の中に存在するか否かしか答えていませんが，この問題が解ければ，W の出現位置をすべて答える問題に拡張することは簡単です．

この問題に対する最も単純な解法は，A の先頭から W を順に探索することです．この方法では，2 重ループによって実装すると $O(NP)$ 時間かかりますし，Knuth Morris Pratt 法のような高速な文字列検索手法を使っても $O(N)$ 時間かかってしまいます．非常に大きな文字列に対して，何度も文字列を検索したいという状況の場合，これでは余りに時間がかかりすぎです．そこで，大きな文字列に対して前処理を施し，検索クエリを高速に処理するということを考えます．

Suffix Array による文字列検索

Suffix Array は，文字列の接尾辞の配列をソートした配列です．Suffix Array といくつかの補助的なデータ構造を使うことで，検索クエリに対して $O(P + \log^2 N)$ の時間で答

えることができます.

準備

文字列 A が与えられたとき，A の i 番目の文字から最後の文字までの部分文字列を，A の i 番目の接尾辞と呼びます．ただし，文字列の最後には，特別な記号$が存在しているとします．例えば，$s =$ “abcabb$” の接尾辞は

- abcabb$
- bcabb$
- cabb$
- abb$
- bb$
- b$
- $

の 7 つです．A の i 番目の接尾辞をここでは Python っぽく $A[i:]$ と表記することにします．また，A の i 番目の文字を $A[i]$ と表記することにします.

Suffix Array は，これらの接尾辞を辞書順にソートしたものです．ただし，$ は他のいかなる文字よりも小さい文字であるとします．上の例の場合，接尾辞配列は次のようになります.

- $
- abb$
- abcabb$
- b$
- bb$
- bcabb$
- cabb$

実際には部分文字列全体をメモリ上に持っておく必要はなく，「何番目の接尾辞であるか」という情報があれば十分です．よって上の例における Suffix Array は “6, 3, 0, 5, 4, 1, 2” と表現できます．これから，A に対する Suffix Array を S で表すことにし，$S[i]$ で i 番目に小さい接尾辞の番号 (つまり，S の i 番目の要素) を表すことにします.

また，文字列 s の先頭の文字から $p-1$ 番目の文字までの部分文字列を s の長さ p の接頭辞と呼び，Python っぽく $s[:p]$ と書くことにします．さらに，二項関係 $<_p$ を長さ p の接頭辞の辞書順で定義します:

$$s <_p t \iff s[:p] < t[:p]$$

$=_p$, $>_p$ なども同様に定義します.

$O(P \log N)$ の方法

大きな文字列 A（長さ N），検索文字列 W（長さ P）とともに A の Suffix Array S が与えられたとき，以下のように S から W を 2 分探索することによって $O(P \log N)$ 時間で A が W を含むかどうかを判定できます．

```
struct Comp {
  const char* A;
  int p;
  Comp(const char* A, int p): A(A), p(p) {}
  bool operator()(int s, const char* w) const {
    return strncmp(A+s, w, p) < 0;
  }
};

bool search1(const char* W, const char* A, const vector<int>& S) {
  int p = strlen(W);
  vector<int>::const_iterator it =
    lower_bound(S.begin(), S.end(), W, Comp(A, p));
  return it != S.end() && strncmp(A+*it, W, p) == 0;
}
```

W が A に存在する必要十分条件は，A が W で始まる接尾辞を持つこと，すなわち，$A[S[i] :] =_P W$ となる i が存在することです．ある k について $A[S[k] :] <_P W$ である場合，条件を満たす i は k より小さく，そうでない場合は，k 以上ですので，これをつかって探索すべき範囲を次々に 1/2 にしていくことができます．

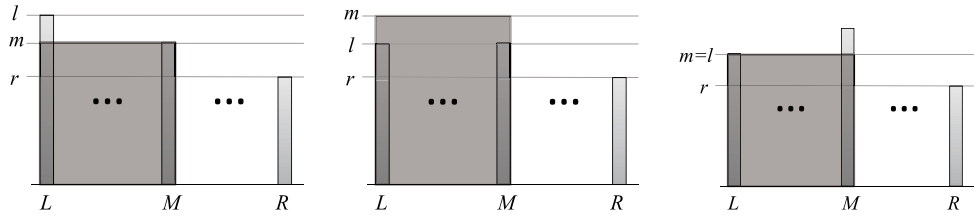
$A[S[i] :] =_P W$ かどうかの判定に $O(P)$ 時間かかり，2 分探索が $O(\log N)$ 回ループするので，全体としてこの解法は $O(P \log N)$ 時間かかります．

$O(P + \log^2 N)$ の方法

$O(P \log N)$ の解法は単純な方法よりも遙かに高速ですが，まだ改善することができます．$O(P \log N)$ の解法では，接尾辞と検索文字列を P 文字比較していましたが，Longest Common Prefix を使うことで比較する文字数を減らすことができます．

文字列 s と文字列 t の Longest Common Prefix $\mathrm{LCP}(s, t)$ は，$s =_p t$ であるような最大の p で定義されます．例えば $s =$ “abcdefg”, $t =$ “abcdge” のとき，$\mathrm{LCP}(s, t) = 4$ です．

2 分探索で $A[S[i] :] <_P W$ かどうかの判定を行うとき，0 文字目から順に比較を行うの



(a) $m < l$　(b) $m > l$　(c) $m = l$

図 1　3 つの場合分け

ではなく，これらの文字列の $\mathrm{LCP}(A[S[i]:], W)$ 文字目を 1 文字見れば，$O(1)$ 時間で比較が完了します．では，$\mathrm{LCP}(A[S[i]:], W)$ を求めるにはどうしたらいいのでしょうか？

今，2 分探索のある時点において，探索範囲が L から R であるとします．また，$l = \mathrm{LCP}(A[S[L]:], W)$ と，$r = \mathrm{LCP}(A[S[R]:], W)$ がわかっているとします．このとき，新しい範囲 L^{new}, R^{new} と，対応する LCP l^{new}, r^{new} を求めることを考えます．一般性を失うことなく $l \geq r$ であると仮定します．ここで，$M = (L + R)/2$, $m = \mathrm{LCP}(A[S[L]:], A[S[M]:])$ とし，以下の 3 通りに場合分けを行います（m の計算方法は後述します）．

(a) $m < l$ である場合
$W =_l A[S[L]:]$ であり，$W \neq_l A[S[M]:]$ であるため，W は必ず範囲の左側にあります．よって，$L^{\mathrm{new}} = L$, $R^{\mathrm{new}} = M$, $l^{\mathrm{new}} = l$, $r^{\mathrm{new}} = m$ となります．

(b) $m > l$ である場合
$A[S[M]] =_{l+1} A[S[L]] \neq_{l+1} W$ であるため，W は必ず範囲の右側あります．よって，$L^{\mathrm{new}} = M$, $R^{\mathrm{new}} = R$, $l^{\mathrm{new}} = l$, $r^{\mathrm{new}} = r$ となります．

(c) $m = l$ である場合
この場合，$\mathrm{LCP}(A[S[M]:], W)$ $(= k$ とおく$)$ はループで直接求める必要があります．ただし，$A[S[M]:] =_l W$ であるため，先頭の l 文字の比較を行う必要はありません．もし，$A[S[M] + k] < W[k]$ である場合，W は必ず範囲の右側にあるため，$L^{\mathrm{new}} = M$, $R^{\mathrm{new}} = R$, $l^{\mathrm{new}} = k$, $r^{\mathrm{new}} = r$ となります．そうでない場合，W は必ず範囲の左側にあるため，$L^{\mathrm{new}} = L$, $R^{\mathrm{new}} = M$, $l^{\mathrm{new}} = l$, $r^{\mathrm{new}} = k$ となります．

この 2 分探索において，l と r はともに非減少であり，(c) でループした回数だけ l か r が増加します．l, r はともに P を超えることはできないので，(c) のループはたかだか $O(P)$ 回しか繰り返されないことがわかります．そして，後述するように m の計算は $O(\log N)$ でできるので，この 2 分探索は $O(P + \log^2 N)$ 時間で実行可能です．

後回しにした $m = \mathrm{LCP}(A[S[L]:], A[S[M]:])$ の計算ですが，次の節ではこの LCP の計算について考えます．

LCP の計算

i 番目に小さい接尾辞 $A[S[i]:]$ と，j 番目に小さい接尾辞 $A[S[j]:]$ の LCP は，Suffix Array 上で隣接する 2 つの接尾辞の LCP の計算に帰着させることができます．

$$\mathrm{LCP}(A[S[i]:], A[S[j]:]) = \min_{i<k\leq j} \mathrm{LCP}(A[S[k-1]:], A[S[k]:])$$

$\mathrm{Height}(k) = \mathrm{LCP}(A[S[k-1]:], A[S[k]:])$ と書くことにすると，この帰着により，Suffix Array の構築に加えて，以下の 2 つができれば，高速な文字列検索ができることになります．

- $\mathrm{Height}(k)$ をすべて求める．
- $\min_{i<k\leq j} \mathrm{Height}(k)$ を高速に求める．

$\mathrm{Height}(k)$ は，LCP の性質をうまく利用することで，Suffix Array から $O(N)$ で構築することができます．また，$\min_{i<k\leq j} \mathrm{Height}(k)$ は，RMQ というデータ構造を構築しておくことで $O(\log N)$ 時間で求めることができます．

以上より，残った問題は 3 つのデータ構造 Suffix Array, LCP, RMQ を予め与えられた巨大な文字列 A から構築するということです．以下の節ではこれらのデータ構造を高速に構築する方法を議論していきます．

Suffix Array の構築

$O(N^2 \log N)$ の方法

Suffix Array は以下のように単純に接尾辞をソートすることで得ることができます．しかしこの方法だと，2 つの接尾辞を比較するのに $O(N)$ 時間かかり，ソートに $O(N \log N)$ 回の比較が必要なので，全体で $O(N^2 \log N)$ 時間かかります．

```
struct Comp1 {
  const char* str;
  Comp1(const char* str): str(str) {}
  bool operator()(int s1, int s2) const {
    return strcmp(str+s1, str+s2) < 0;
  }
};

void build_suffix_array1(const char* A, /*OUT*/ vector<int>& S) {
  int n = strlen(A);
```

```
  S.resize(n+1);
  for (int i = 0; i < n+1; ++i) { S[i] = i; }
  sort(S.begin(), S.end(), Comp1(A));
}
```

実は Suffix Array は $O(N)$ 時間で構築できることが知られています．しかし，Suffix Array を $O(N)$ 時間で構築するアルゴリズムは難しいので，ここでは $O(N \log^2 N)$ で構築する方法を紹介します．

$O(N \log^2 N)$ の方法

長さ h の接頭辞で比較した順序を h-order と呼ぶことにします．もし，ある h に対して $A[S[i]:] <_h A[S[j]:]$ であれば，$A[S[i]:] < A[S[j]:]$ であり，また $A[S[i]:] >_h A[S[j]:]$ であれば，$A[S[i]:] < A[S[j]:]$ です．この性質を使うと，文字列 A が与えられたときに，1-order, 2-order, 4-order, 8-order, ... と順々にソートしていくことで，$O(N \log^2 N)$ で Suffix Array を構築できます．

まず，1-order でのソートは簡単に実行できます．

次に，h-order でのソートができているとき，$2h$-order でソートすることを考えます．$A[S[i]:] \neq_h A[S[j]:]$ であるような $S[i]$, $S[j]$ の組は上の議論から既に順序が確定しているので，$A[S[i]:] =_h A[S[j]:]$ となっているグループをそれぞれソートすればよいことがわかります．$A[S[i]:] =_h A[S[j]:]$ であるような $S[i]$, $S[j]$ に対して $2h$-order をつけるには，h 文字目から $2h-1$ 文字目までの文字列を比較すればよいのですが，実はこの比較は既に h-order でソートした時点でわかっています．すなわち，この $S[i]$, $S[j]$ の $2h$-order における順序は，h-order における $S[i+h]$, $S[j+h]$ の順序に一致します．これで，h-order でのソートができたとき，$2h$-order でソートすることができるようになりました．

比較のコストが $O(1)$ であり，ひとつの h に対してソートを行うのに $O(N \log N)$ 回の比較が必要であり，h は $O(\log N)$ 個あるので，Suffix Array の構築にかかる時間は全体で $O(N \log^2 N)$ です．

Suffix Array 構築の実装

ここでは以下のような実装をします．

- 典型的なインプットでは，1-order でのソートに時間が結構かかるので，1-order でのソートはバケツソートで行います．
- 各接尾辞の順序関係を group という配列で管理します．group[i] は，ソートの各段階において， $A[S[i]:] =_h A[S[j]:]$ であるような最小の j を保存しています．
- 「次」のグループを指す next という配列を持っておきます．next[i] は

group[array[j]] > group[array[i]] であるような最小の j を保存しています．ただし，他のいかなる next[j] からも指されていないような i における next[i] の値は Don't Care ということにします．

- もし，あるグループ内に 1 つの要素しかない場合，そのグループはソートする必要がありません．もし，次のグループがソートする必要のないグループなら next[i] の符号を負にしておきます．ソートする必要のないグループが複数個連続して存在している場合，それらをまとめてスキップできるように next[i] の値を調整していきます．
- ソートした結果で group を更新する必要があるのですが，ソート中に group を上書きしてしまうとそれ以降のソートがうまくいかないので，まずは next だけを更新し，後で next を見ながら group を更新します．

```
struct Comp2 {
  int h;
  const vector<int>& group;
  Comp2(int h, const vector<int>& group): h(h), group(group) {}
  bool operator()(int s1, int s2) const {
    return group[s1+h] < group[s2+h];
  }
};

void build_suffix_array2(const char* A, /*OUT*/ vector<int>& S) {
  typedef unsigned char uchar;
  int n = strlen(A);
  vector<int> group(n+1), next(n+2), bucket(256);
  S.resize(n+1);
  for (int i = 0; i < n+1; ++i) { ++bucket[(uchar)A[i]]; }
  for (int i = 1; i < 256; ++i) { bucket[i] += bucket[i-1]; }
  for (int i = 0; i < n+1; ++i) { S[--bucket[(uchar)A[i]]] = i;  }
  for (int i = 0; i < n+1; ++i) { group[i] = bucket[(uchar)A[i]]; }
  for (int i = 0; i < 255; ++i) { next[bucket[i]] = bucket[i+1]; }
  next[bucket[255]] = n+1;

  for (int h = 1; next[0] != -n-1; h *= 2) {
    int prev = -1;
    for (int i = 0; i < n+1; i = abs(next[i])) {
      if (next[i]-i <= 1) {
        if (prev == -1) { prev = i; }
```

```
        else { next[prev] = -i; }
      } else {
        prev = -1;
      }
    }
    if (prev != -1) { next[prev] = -n-1; }

    Comp2 comp(h, group);
    for (int i = 0; i < n+1; ) {
      if (next[i] < 0) {
        i = -next[i];
      } else {
        sort(S.begin()+i, S.begin()+next[i], comp);
        int prev = i, nxt = next[i];
        for (int j = i+1; j < nxt; ++j) {
          if (comp(S[j-1], S[j])) { next[prev] = j; prev = j; }
        }
        i = next[prev] = nxt;
      }
    }
    for (int i = 0; i < n+1; i = abs(next[i])) {
      for (int j = i; j < next[i]; ++j) { group[S[j]] = i; }
    }
  }
}
```

Longest Common Prefix の構築

以下のように $i-1$ 番目の文字列と i 番目の文字列を前から比較していくことで Height(i) を構築できますが，この方法だと全体で $O(N^2)$ 時間かかってしまいます.

```
void build_lcp1(const char* A, const vector<int>& S,
                /*OUT*/ vector<int>& height) {
  int n = S.size() - 1;
  height = vector<int>(n+1);
  for (int i = 1; i < n+1; ++i) {
    while (A[S[i-1]+height[i]] == A[S[i]+height[i]]) { ++height[i]; }
```

```
  }
}
```

しかし，うまい順番で Height(i) を計算していき，無駄な繰り返しを避けることで $O(N)$ で Height(i) を全部計算することができます．

ここで，Rank(i) で，i 番目の接尾辞が何番目に小さいかを表すことにします．すなわち，Rank($S[i]$) $= i$ です．

(1) LCP($A[S[i]:], A[S[j]:]$) $= \min_{i<k\leq j}$ LCP(A[S[k $-$ 1] :], A[S[k] :]) より $i < k \leq j$ に対して LCP($A[S[k-1]:], A[S[k]:]$) $\leq$ LCP($A[S[i]:], A[S[j]:]$) が成立します．

(2) LCP($A[S[i-1]:], A[S[i]:]$) > 1 のとき，最初の文字を削除しても接尾辞の順番は保存されるので Rank($S[i-1]+1$) $<$ Rank($S[i]+1$) が成立します．

(3) LCP($A[S[i-1]:], A[S[i]:]$) > 1 のとき，最初の文字を削除すると LCP が 1 だけ減少するので LCP($A[S[i-1]+1:], A[S[i]+1:]$) $=$ LCP($A[S[i-1]:], A[S[i]:]$) -1 が成立します．

Height(Rank($i-1$)) $=$ LCP($A[S[$Rank($i-1$) $-1]:], A[S[$Rank($i-1$)$]:]$) > 1 のとき，(2) より Rank($S[$Rank($i-1$) $-1]+1$) $<$ Rank($S[$Rank($i-1$)$]+1$) $=$ Rank(i) が成り立ちます．したがって，Rank($S[$Rank($i-1$) $-1]+1$) $\leq$ Rank(i) -1 となります．よって (1) より

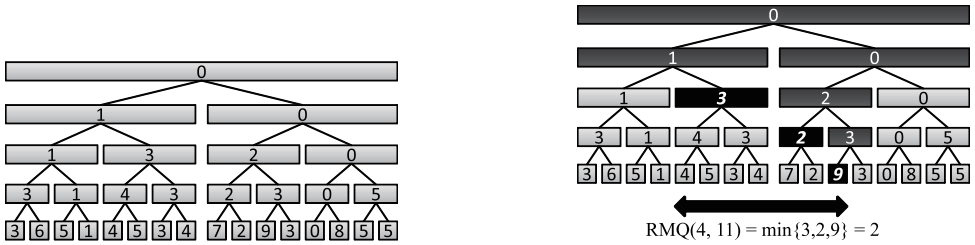
$$\begin{aligned}\text{Height}(\text{Rank}(i)) &= \text{LCP}(A[S[\text{Rank}(i)-1]:], A[S[\text{Rank}(i)]:]) \\ &= \text{LCP}(A[S[\text{Rank}(i)-1]:], A[i:]) \\ &\geq \text{LCP}(A[S[\text{Rank}(i-1)-1]+1:], A[i:])\end{aligned}$$

ゆえに (3) より

$$\begin{aligned}\text{Height}(\text{Rank}(i)) &\geq \text{LCP}(A[S[\text{Rank}(i-1)-1]:], A[i-1:]) - 1 \\ &= \text{Height}(\text{Rank}(i-1)) - 1\end{aligned}$$

したがって，Rank(i) の順番で Height を計算していけば，Height の最大値が $O(N)$ であり，-1 される回数も高々 $O(N)$ 回なので，全体として $O(N)$ 回の繰り返しで Height を構築することができます．

```
void build_lcp2(const char* A, const vector<int>& S,
                /*OUT*/ vector<int>& height) {
  int n = S.size() - 1, h = 0;
  vector<int> rank(n+1);
  height = vector<int>(n+1);
  for (int i = 0; i < n+1; ++i) { rank[S[i]] = i; }
  for (int i = 0; i < n+1; ++i) {
    if (rank[i] > 0) {
```



(a) ノードの幅が区間 $[L_v, R_v)$ を表し，ノードの値は M_v を表す.

(b) クエリに対する処理

図 2　Segment Tree

```
      int j = S[rank[i]-1];
      while (A[i+h] == A[j+h]) { ++h; }
      height[rank[i]] = h;
    }
    if (h > 0) { --h; }
  }
}
```

Range Minimum Query

この節では最後に残った RMQ について議論します.

次のような問題を考えます.

> 長さ n の配列 $\{a_i\}$ が与えられる．この配列に対して Range Minimum Query が大量に与えられる．Range Minimum Query は以下の形式を取る.
>
> $$\mathrm{RMQ}(L, R) = \min_{L \le i < R} \{a_i\}$$
>
> Range Minimum Query にできるだけ高速に答えよ.

Segment Tree を使った解法

配列 $\{a_i\}$ に対して，$O(n)$ 時間かけて Segment Tree を構築することで，一つのクエリに $O(\log n)$ 時間で答えることができます.

簡単のため，以下では n は 2 の冪であるとします．もし，n が 2 の冪でないときは，配列の末尾に十分小さな値を追加することで，配列のサイズを 2 の冪にできるので，これは本質的な制約ではありません.

ここで使用する Segment Tree は次の条件を満たす 2 分木です.

- Segment Tree の各ノード v は，半開区間 $[L_v, R_v)$ 及びその区間における最小値 $M_v = \min_{L_v \le i < R_v} \{a_i\}$ を持つ．
- 根ノード r は，半開区間 $[0, n)$ を持つ．
- 各ノード v は，$L_v + 1 \neq R_v$ のとき，ちょうど 2 つの子ノードを持つ（子ノードがちょうど 2 つなのは，n が 2 の冪であることによる）．
- ノード v の子ノード w_1，w_2 は，それぞれ半開区間 $\left[L_v, \frac{L_v+R_v}{2}\right)$，半開区間 $\left[\frac{L_v+R_v}{2}, R_v\right)$ を持つ．

このような木が構築できたとして，この木を使って RMQ に $O(\log n)$ 時間で答えることを考えます．

まず，以下の部分問題を考えます．

> RMQ(v, L, R): Segment Tree のノード v と，クエリ区間 $[L, R)$ が与えられる．ただし，$[L, R) \subseteq [L_v, R_v)$ である．このとき，RMQ(L, R) を答えよ．

まず，$[L, R) = \emptyset$ であるときは，∞ が解であり，また，$[L_v, R_v) = [L, R)$ のときは，M_v が解です．そうでない場合は，$[L, R)$ を 2 つの区間 $[L_1, R_1) = \left[L, \min\left\{R, \frac{L_v+R_v}{2}\right\}\right)$，$[L_2, R_2) = \left[\max\left\{L, \frac{L_v+R_v}{2}\right\}, R\right)$ に分割し，そして，v の子ノード w_1，w_2 に対して，それぞれ RMQ(w_1, L_1, R_1)，RMQ(w_2, L_2, R_2) を解き，それらの min を取ることで，RMQ(v, L, R) の解を得ことができます．

部分問題の解法が分かったので，RMQ(L, R) に対する解は，Segment Tree の根ノードを r として，RMQ(r, L, R) を計算することで求めることができます．実装は以下のようになります（実装では木を 1 次元の配列に格納しています．i 番目のノードの 2 つの子は $2i + 1$ と $2i + 2$ 番目に格納されています）．

```
int rmq_min(const vector<int>& rmq, int v,
            int Lv, int Rv, int Lq, int Rq) {
  int mid = (Lv+Rv)/2, ret = INT_MAX;
  if (Rq < mid) { return rmq_min(rmq, 2*v+1, Lv, mid, Lq, Rq); }
  if (Lq > mid) { return rmq_min(rmq, 2*v+2, mid, Rv, Lq, Rq); }
  if (Lq < mid) {
    if (Lq == Lv) { ret = min(ret, rmq[2*v+1]); }
    else { ret = min(ret, rmq_min(rmq, 2*v+1, Lv, mid, Lq, mid)); }
  }
  if (Rq > mid) {
    if (Rq == Rv) { ret = min(ret, rmq[2*v+2]); }
    else { ret = min(ret, rmq_min(rmq, 2*v+2, mid, Rv, mid, Rq)); }
  }
  return ret;
```

```
}

int rmq_min(const vector<int>& rmq, int L, int R) {
  int n = (rmq.size()+1)/2;
  if (L >= R) { return INT_MAX; }
  if (L == 0 && R == n) { return rmq[0]; }
  return rmq_min(rmq, 0, 0, n, L, R);
}
```

RMQ(L, R) に答えるのに必要な時間は $O(\log n)$ です．これは次の理由によります．区間を分割したときに生じる 2 つの区間について，次の 3 つの場合があります

(a) 片方の区間がクエリ区間を全く含まない．
(b) 片方の区間がクエリ区間に完全に含まれる．
(c) (a), (b) のいずれでもない．

まず，(c) は高々 1 回しか起きません．一度 (c) が起こると，それ以降の分割はすべて (b) になるからです．さらに，(a), (b) の場合，2 つの部分問題のうち片方は自明な問題であり，実質的に解く必要があるのは，もう片方の問題だけです．したがって，RMQ(L, R) に答えるために解く必要のある部分問題の数は，Segment Tree の高さ，すなわち $O(\log n)$ に比例します．

次に，Segment Tree を構築することを考えます．まず，葉ノード v に関しては，区間のサイズが 1 なので，$M_v = a_{L_v}$ とすれば OK です．葉でないノード v は，2 つの子ノード w_1, w_2 の値の min を取ります．

$$M_v = \min\{M_{w_1}, M_{w_2}\}$$

Segment Tree の構築はこれだけです．

```
void build_rmq(const vector<int>& array,
               /*OUT*/ vector<int>& rmq) {
  int n = array.size(), N = 1;
  while (N <= n) { N *= 2; }
  rmq = vector<int>(2*N-1, INT_MAX);
  copy(array.begin(), array.end(), rmq.begin()+N-1);
  for (int v = N-2; v >= 0; --v) {
    rmq[v] = min(rmq[2*v+1], rmq[2*v+2]);
  }
}
```

検索再び

以上で，巨大な文字列に対して前処理（Suffix Array，LCP および RMQ の構築）を行い，与えられた部分文字列が含まれているかどうかの判定を高速に行えるようになりました．まず，以下のように前処理します．

```
vector<int> S, height, rmq;
build_suffix_array2(A, S);
build_lcp2(A, S, height);
build_rmq(height, rmq);
```

これらの前処理したデータを使って文字列検索を実装すると以下のようになります．

```
inline int suffix_suffix_lcp(int i, int j, const vector<int>& rmq) {
  return rmq_min(rmq, i+1, j+1);
}

inline int suffix_string_lcp(int s_i, int p, const char* W,
                                      int n, const char* A) {
  int j = 0;
  while (j < p && A[s_i + j] == W[j]) { ++j; }
  return j;
}

bool search2(const char* W, const char* A,
             const vector<int>& S, const vector<int>& rmq) {
  int p = strlen(W), n = S.size()-1;
  int L = 0, R = n;
  int l = suffix_string_lcp(S[L], p, W, n, A);
  int r = suffix_string_lcp(S[R], p, W, n, A);
  while (R - L > 1) {
    int M = (L+R)/2;
    if (l >= r) {
      int m = suffix_suffix_lcp(L, M, rmq);
      if (m < l) { R = M; r = m; }
      else if (m > l) { L = M; }
      else {
        int k = l + suffix_string_lcp(S[M]+l, p-l, W+l, n, A);
```

```
        if (k == p || A[S[M]+k] < W[k]) { L = M; l = k; }
        else { R = M; r = k; }
      }
    } else {
      int m = suffix_suffix_lcp(M, R, rmq);
      if (m < r) { L = M; l = m; }
      else if (m > r) { R = M; }
      else {
        int k = r + suffix_string_lcp(S[M]+r, p-r, W+r, n, A);
        if (k == p || A[S[M]+k] < W[k]) { L = M; l = k; }
        else { R = M; r = k; }
      }
    }
  }
  return l == p || r == p;
}
```

Suffix Array から部分文字列を 2 分探索している部分を少し変更すると，部分文字列が出現するすべての場所を取得するといったことも可能です．Suffix Array や RMQ を構築する方法は，ここにあげた方法以外にもいくつか存在するので，興味がある人は調べてみるとおもしろいと思います．

参考文献

1. Udi Manber, Gene Myers: *Suffix Arrays — A New Method for On-line String Searches*
2. Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikaw, and Kunsoo Park: *Linear-Time Longest-Common-Prefix Computation in Suffixx Arrays and Its Applications*
3. N. Jesper Larsson, Kunihiko Sadakane: *Faster Suffix Sorting*
4. 岡野原 大輔:*Suffix Array* (`http://homepage3.nifty.com/DO/suffix_array.htm`)
5. 前原 貴憲:*Spaghetti Source* (`http://www.prefield.com/algorithm/`)

最大流問題とフロー分解定理

COS

はじめに

このコラムでは，ICPC などのプログラミングコンテストを念頭において，グラフ理論のアルゴリズムとしてよく知られている最大流問題（ネットワークフロー問題）について簡単な解説を行う．まず初めに最大流問題の概略を説明し，解法を述べる．次にアルゴリズムの肝となる逆辺の意味とフロー分解定理を簡単に解説する．最後にフロー分解定理の実際の使用例を述べる．

最大流問題の概略

最大流問題を定義するために頂点と辺がある有向グラフ G に対して容量関数 $u : E(G) \to \mathbb{R}_+$[*1]と関数 $f : E(G) \to \mathbb{R}_+$ を付け加える．関数 $f(e)$ がすべての $e \in E(G)$ で $f(e) \leq u(e)$ を満たすときにフロー f と呼ぶ．フロー f の点 v での正味流入量 $ex_f(v)$ は

$$ex_f(v) := \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$$[*2]

として定義する．$ex_f(v) = 0$ ならば f は v でフロー保存則を満たしていると呼ぶ．与えられたネットワーク (G, u, s, t) で $ex_f(s) = -ex_f(t) \leq 0$ であり，かつ $v \in V(G) \backslash s, t$[*3] で $ex_f(v) = 0$ であるときフロー f は s-t-フローと呼ぶ．$ex_f(t)$ が最大の s-t-フローを求めるのが最大流問題である．

数式を並べたが簡単に言うと，有向グラフの各辺に容量がついているときに s から t へどの程度水を流せるかという問題である．では次の節から現実的な時間で最大流問題を解く手法を説明する．

[*1] $E(G)$ はグラフ G の辺を表す．
[*2] $\delta^-(v)$ は v へ入る辺，$\delta^+(v)$ は v から出る辺．
[*3] s,t 以外のグラフ G の頂点集合．

最大流問題の解法

最大流問題を解くために逆辺と残余グラフという概念を新しく考える．有向グラフの各辺 $e = (v, w)$ に対して w から v へ向かう新しい辺 $\overleftarrow{e}$ を考える．$\overleftarrow{e}$ を G に加えたグラフを

$$\overleftrightarrow{G} := (V(G), E(G) \dot{\cup} \{\overleftarrow{e} : e \in E(G)\})$$

とする．$\overleftarrow{e}$ を e の**逆辺**と呼ぶ．

次にフロー f に対する容量関数を考える．これは残容量関数と呼び，$u_f(e) := u(e) - f(e)$, $u_f(\overleftarrow{e}) := f(e)$ と定義する．さらにこの残容量関数 u_f を用いて定義されるグラフ

$$G_f := (V(G), \{e \in E(\overleftrightarrow{G}) : u_f(e) > 0\})$$

を**残余グラフ**と呼ぶ．残余グラフはフロー f を流した後にまだ使える辺と使えるようになった逆辺を集めたグラフと解釈すれば良い．残余グラフ上に存在する s,t パス（または任意の閉路）に沿ってフロー f を γ 増やすことを**水を流す**と表現することにする．

さてこの残余グラフ上で s,t パスが存在しないとするとフロー f は明らかに最適な s-t-フローとなっている．そうでない場合，つまり s,t パスが存在する場合はそのパスにそって水を流せば明らかに s-t-フローは増加する．s-t-フローを増加させた後に新しい残余グラフをつくれば帰納的に最適な s-t-フローが求まる．これが **Ford-Fullkerson アルゴリズム**である．

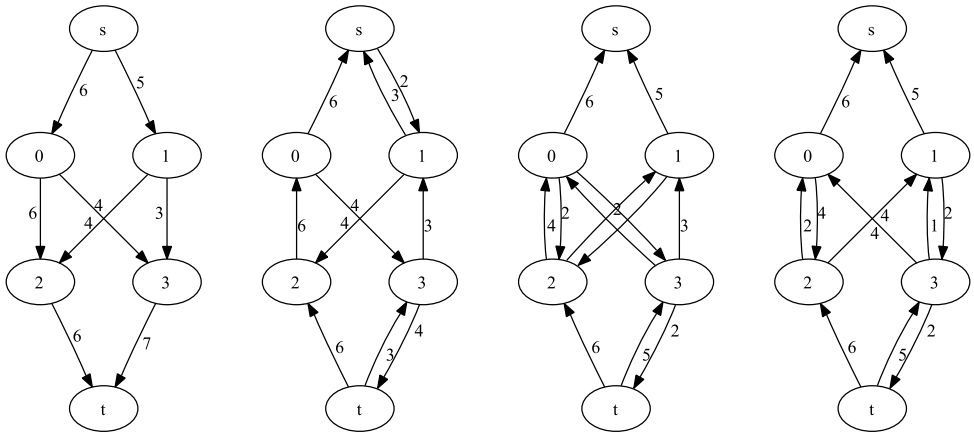
Ford-Fullkerson アルゴリズムは s,t パスの求め方によっては多項式時間で計算が終了しない場合がある[*4]．ではどのように s,t パスを求めれば良いのであろうか？答えは簡単で幅優先探索を用いて s,t パスを求めれば $O(m^2n)$[*5]で最大流問題を解くことができる．この s,t パスを幅優先探索を用いて探すアルゴリズムを **Edmonds-Karp アルゴリズム**と呼ぶ．s,t パスの求め方をさらに工夫して，レベルグラフを作成し深さ優先探索を使用する **Dinic のアルゴリズム**は $O(n^2m)$ で最大流問題を解く．

逆辺の意味とフロー分解定理

前節で述べた逆辺とはアルゴリズム上でどのような役割を持っているのだろうか？グラフの話なのに図が無いと分かり辛いのでそろそろ図を用いて解説を行う．図 1(a) のような 6 頂点のグラフを考える．このグラフ上で $s \to 0 \to 2 \to t$ と $s \to 1 \to 3 \to t$ に流せるだけ水を流すと残余グラフは図 1(b) になる．逆辺の概念が無いとすると，つまり $(2, 0), (3, 1)$ の辺が無い場合，これ以上水が流せず，最適な s-t-フローが見つかるかどうかは s,t パスの選び方に依存してしまう．しかし，逆辺があると $s \to 1 \to 2 \to 0 \to 3 \to t$

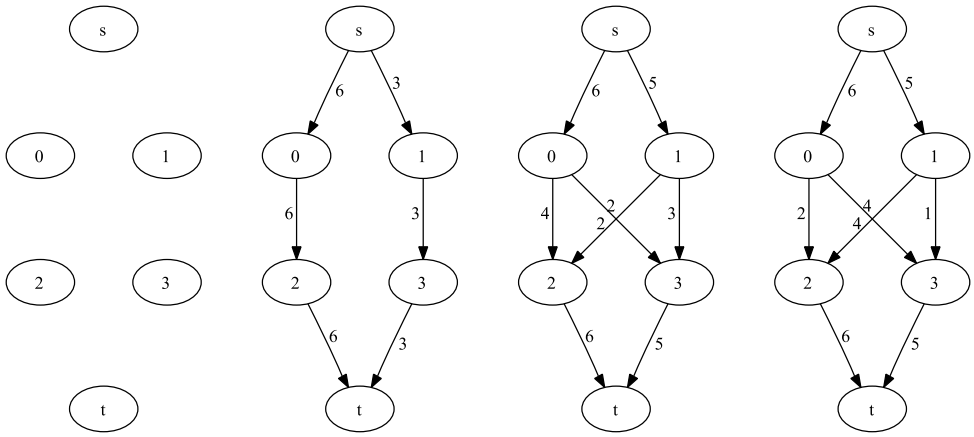
[*4] Ford-Fullkerson アルゴリズムでは s,t パスの求め方を規定してない．

[*5] n は頂点の数，m は辺の数．



(a) 初期状態　(b) 途中経過　(c) 最適な場合 1　(d) 最適な場合 2

図 1　元のグラフと水を流した後の残余グラフ



(a) 初期状態　(b) 途中経過　(c) 最適な場合 1　(d) 最適な場合 2

図 2　各状態でのフロー

と水を流すことが出来，残余グラフは図 1(c) となる．この残余グラフ上にはもう s,t パスは無いのでこれが最適な s-t-フローである．ここで途中経過のフロー（図 2(b)）と最適な場合 1 のフロー（図 2(c)）を見比べると，$(0,2)$ の辺に流れるフローが 6 から 4 に減っていて，代わりに $(0,3)$ に減った分のフローが流れていることが分かる．つまり，逆辺 $(2,0)$ の部分で $(0,2)$ に流れていた水がキャンセルされてフローの向きが変わる．このように逆辺（最大流問題）は，今まで行ってきた処理に対してバックトラックをするような機能があるのだ．さらによく見てみると，$s \to 1 \to 2 \to 0 \to 3 \to t$ というパスは $s \to 0 \to 2 \to t$ のパス（の流量 2 個分）と組み合わせて流量 2 のパス $s \to 1 \to 2 \to t$, $s \to 0 \to 3 \to t$ と流量 2 の閉路 $0 \to 2 \to 0$ の 3 つに分解されていることが分かる．

さて，図 1(c) の状態で s,t への水を流すことはできない．しかし他の部分ではどうであ

ろうか？例えば $1 \to 2 \to 0 \to 3 \to 1$ の閉路に水を流すとどうなるであろうか？結果は図 2(d) の様にフローは流れ，残余グラフは図 1(d) になる．このように閉路にも水を流すことが出来てその結果別の s-t-フローを得ることができる．**フロー分解定理**という物があるがこの定理は任意の s-t-フローは s,t パスと閉路の集合に分解できるということ言っている定理であり，この $1 \to 2 \to 0 \to 3 \to 1$ の閉路に水を流すことはこの定理の一例である．

実際の使用例

フロー分解定理とは実際にどのような場面で使うのであろうか？ここでは実際の問題を見ながら解説していこう． 問題として ICPC の North America 2010 で出題された Data Recovery[*6]を使用する．

まず，問題を簡単に解説する．表 1 のような各マスに $[0, 100]$ の整数もしくは空欄が書かれた表が与えられる．さらに，各行・各列の総和も与えられる．空欄の箇所には $[0, 100]$ の整数を入れることができるのだが，一意に値を定めることができるならその空欄を埋めよという問題である．表 1 に対する答えは表 2 になる．なお表の最大サイズは最大 50×50 である．

表 1　入力

	18	6	17	15	17
21			6		8
10			0	4	2
15	3			5	
13	4	0	2		
14	2	1	5		

表 2　答え

	18	6	17	15	17
21			6	0	8
10			0	4	2
15	3	3	4	5	0
13	4	0	2		
14	2	1	5		

さてこの問題はどうやって解けばよいのであろうか？中央の行の 4 はまだしも 3 と 0 はどうやれば決定できるのだろうか？答えは最大流の記事なのだからもちろん最大流を用いて解くことができる．ではそのアルゴリズムを順に述べていくことにする．

まず，総和からすでに判明してる数値を引いておく．次に図 3 のように s ノード，X ノード，Y ノード，t ノードからなる 2 部グラフを作成する．s ノードから Y ノードへの辺の容量は各列の総和，X ノードから t ノードへの辺の容量は各行の総和とする．Y ノードから X ノードへの辺は (y, x) マスが空欄の時にかぎり容量が 100 の辺を張る．このグラフ上で最大流を流すと表の条件を満たす解が一つ得られる．あとは各マスについてその解が一意かどうかを調べれば良いだけになった．

どうやれば解が一意かどうかを調べることができるのだろうか？アイディアとしては

[*6] http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=4865

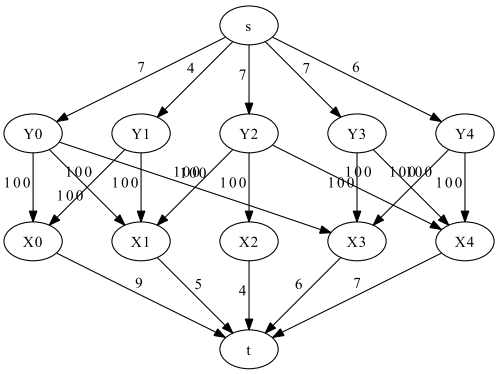


図 3　s，Y ノード，X ノード，t からなるグラフ

(x, y) の辺に求めた解より多いもしくは少ない水を流した状態で最大流が成立するかどうかを調べれば良い．制約を課した最大流 (Dinic) を何度も使えばできるであろうがそれでは計算量が $O(100^6)$ 程度になって間に合わない．最大流を何度も使用するのはまずいことが分かったがどうすれば高速化できるのだろうか？

ここでフロー分解定理が登場する．フロー分解定理を用いるとある程度自由に最大流の水の流れを変えることができるので，それを用いて答えが一意に定まるかどうかチェックすれば良い．具体的には残余グラフ上で各 (y, x) の辺と逆辺の (x, y) に対してその辺を含む閉路があるかどうかを調べる．もし閉路が存在すればそのマスの値は変えることができるので答えは一意に定まらず，存在しなければ答えは一意に定まる．これの計算量は $O(100^4)$ 程度なので時間的にも問題ないであろう．解答のソースコードは 200 行ほどありここに掲載するには長いのでブログ[*7]に載せてある．

このようにフロー分解定理は最大流を何度も行いたい場合に計算量を抑えてフローの形を変えることができるという便利な物である．

まとめ

このコラムでは逆辺を重視して最大流問題の簡単な解説を書いた．実際の所 Dinic が書けて，逆辺の意味が理解でき，MinCut，2 部グラフのマッチングとフロー分解定理ができればプログラミングコンテストに出題される最大流問題はほぼ解けるであろう．腕に自身のある方は ICPC の Bangladesh 2010 で出題された Hyper Knights[*8]を解くのをおすすめする．上記の内容をかなり含んでいる非常に難しい問題である．最大流問題に関しての別のアルゴリズムや厳密な議論，例えば計算量の解析，などが知りたい人は『組合せ最適化』[*9]を読むのがおすすめである．

[*7] http://d.hatena.ne.jp/cos65535/20101130/1291092835

[*8] http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=4862

[*9] (2009/03)『組合せ最適化―理論とアルゴリズム』シュプリンガージャパン; 第 2 版 本コラムの参考文献でもある．

拡張現実で遊ぼう

tsutcho

はじめに

拡張現実とは

そもそも拡張現実 (AR, Augumented Reality) とは，現実の環境にコンピュータで情報を付加すること，その付加された状況を意味する言葉である．本来ならば視覚に限定されてはいないのだが，実際には視覚に関するものを指すことが非常に多い．

拡張現実の歴史

AR は新しい技術のように思えるが，1965 年，ハーバード大学のアイヴァン・サザーランド氏が HMD（ヘッドマウントディスプレイ）を用い，CG を現実世界上に重ねて表示したことが始まりである．では，なぜその技術が最近になって注目を集めているのだろうか．その理由は 2 つある．

- フレームワークの充実
- モバイル端末の普及

それぞれ AR アプリを「作る」側と「使う」側にとっての革新である．ここでは，そんな AR の現在を「作る」「使う」という 2 つの観点で見ていこうと思う．

拡張現実を作ろう

まずは AR アプリの開発についての話である．と言っても画像認識などといった難しいことは無しに既存のフレームワークを使う．そういった知識が一切無くても作れてしまうようなフレームワークが多数存在するためである．

ARtoolkit[*1]

現在最も有名な AR のフレームワークであろう．奈良先端科学技術大学院大学の加藤博一教授とワシントン大学 HITL によって開発された．その扱いやすさと NyARtoolkit[*2]などといった派生系の登場により，広く用いられている．Android に対応した派生版もあり，「AR」といえばこのマーカーを思い出す人も多いのではないだろうか．図 1 は実際に筆者が撮影したものである．NyARtoolkit を用い，書いたコードはモデルの読み込みと表示，あとは使用マーカーの選択，サイズの指定程度である．マーカーのデータの抽出も自動で行

[*1] http://www.hitl.washington.edu/artoolkit/

[*2] http://nyatla.jp/nyartoolkit/wiki/index.php

われ，モデルの表示も通常と同様である．これならば 3D 空間にモデルを表示することができるならば誰でも AR アプリケーションの開発が可能だろう．

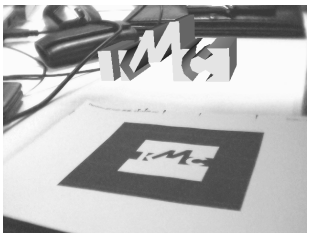


図 1　ARtoolkit の使用例．自作．

PTAM[*3]

PTAM (Parallel Tracking and Mapping) はオックスフォード大学の Georg Klein 氏らによって開発されたマーカレス AR である．マーカー無しで任意の場所から空間を生成することができる．ARtoolkit のように誰でも使えるほど手軽ではないが，マーカーが不要というのは大きな特徴である．マーカーを使用しないということは，画面に不自然なマーカーを写りこませる必要がないという長所であると同時に，空間の座標を指定できないという短所でもある．どちらが優れているということはなく，求めるものによって使い分けるといいだろう．

拡張現実を使おう

続いて，AR を「使う」立場としての話である．PC を用いた AR にはカメラの購入，場所の限定という制約が存在するため，一般ユーザー向けの AR はスマートフォンなどを用いたものが多い．ということでスマートフォンなどのモバイル端末向け AR についての話を主にする．

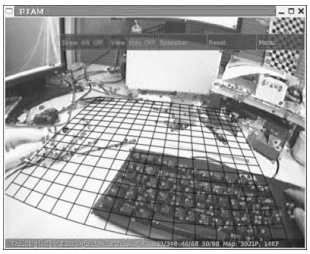


図 2　PTAM の使用例．公式サイトより

セカイカメラ[*4]

スマートフォン向け AR アプリで最も話題になったものといえば，セカイカメラであろう．このセカイカメラとは，カメラで撮影した画像に「エアタグ」と呼ばれる情報を重ねて表示するアプリである．このエアタグはユーザーが自由に付与することができる．有用な情報を求めるものではなく，単純に空中に浮かび上がるエアタグを見て楽しむという用途で用いられることが多い．一時期非常に話題になったが現在では下火である．その性質上有用な情報が少なく，かと言ってコンテンツとしても「AR である」ということ以上の物がなかったためだろう．

図 3　セカイカメラの例．公式サイトより

[*3] http://www.robots.ox.ac.uk/~gk/PTAM/

[*4] http://sekaicamera.com/

商業利用

スマートフォンの普及に伴い，スマートフォンを使うことを前提としたARを用いたキャンペーンを行う企業が増えている．「ラブプラス」ではカノジョがマーカー上に登場したりエヴァンゲリオンが現れたり，デニーズでは声優が現れメニューを解説したりと使い方は様々である．しかし，現時点ではスマートフォンの所持者が限られることから，幅広い層へのアピールは期待できない．逆に言えば，スマートフォンを持っているような人をターゲットとしたキャンペーンを打つ必要があるということである．声優を起用したデニーズが好例である．スマートフォンユーザーが増加するこれからならばさらに注目を集めるかもしれない．

拡張現実のこれから

ここまでは現在までのARについての話であった．では，ARの未来に関する話を2つ紹介しよう．

HMD

ARについて語る際，よく例として用いられるのが電脳コイルの「電脳メガネ」やドラゴンボールの「スカウター」と言った，メガネ型の物に投影されるタイプのものである．このようにメガネのような形で装着するディスプレイをHMD（ヘッドマウントディスプレイ）と呼ぶ．かつてはHMDと言っても映画などを臨場感あふれる中で見ることを目的とした外の見えないものばかりであった．見た目も非常に大きく，つけて外に出られるものではなかった．近年ではブラザー社のAiRScouter[*5]のようにレーザーを直接角膜に照射することで本来の視界を損なわずに使えるHMDが開発され，発売が予定されている．これは非常に小型でメガネに装着して用いることができ，外から見てもさほど違和感ない作りとなっている．これの発売により，HMDを付けた人間が町を闊歩する時代がやってくるかもしれない．

図4　ブラザー社が発売予定のHMD，AiRScouter．公式サイトより

SmartAR[*6]

SmartARは現在注目を集めているSONYが開発中のモバイル端末向けAR技術である．注目すべきはその性能の高さである．まず任意の画像をマーカーとして用いることができる．ARのマーカーといえばそれがマーカーであることを示す枠などが付いていることが多く，それが不要になることの影響は大きい．例えば壁に貼ってあるチラシを撮影することで追加の情報を得たり，もともと存在する看板や絵画などに情報を付加したりといった活用が可能となる．さらに，マーカーが画面に写って

[*5] http://www.brother.co.jp/news/2010/airscouter/index.htm

[*6] http://www.sony.co.jp/SonyInfo/News/Press/201105/11-058/

いなくともそれまでのマーカーと視点の移動から位置を把握し続けることができる．これは AIBO のための技術を利用して生み出された機能であるらしい．これによりマーカー型の長所と非マーカー型の長所の双方を活かすことが可能となるのである．

図 5　SmartAR の例．ただの絵がマーカーになっている．Sony の公式サイトより

最後に

まとめ

40 年以上前に生み出された AR という技術は，今や漫画の中のものではなく，私達でも簡単に楽しむことができるものとなっている．これからさらなる技術革新により，AR はもっと身近になるだろう．そして完全に生活の一部になる時がやってくるかもしれない．拡張現実が「現実」になる世界はまさに「未来」と呼ぶにふさわしいものとなるだろう．

LIBLINEAR で実践機械学習入門

tahi

はじめに

自己紹介

こんにちは，京都大学工学部電気電子工学科に所属している tahi といいます．普段は電気や電子や工学について勉強して，実験ではモーターを回したりトランジスタを焦がしたり電波を出したりしているのですが，今回は主に趣味でやっている内容として，プログラムを書いて LIBLINEAR で遊んでみよう，という感じの記事を書きたいと思います．できるだけ初心者向けに丁寧に書いていきたいのですが，紙面の都合などもあるのでサンプルソースなどは載せず Web ページで補足，という形にしたいと思います．`http://www.kmc.gr.jp/~tahi/C80liblinear.html` に補足をいろいろと載せておくので良ければ参考にしてください．

LIBLINEAR ってなに？

説明しよう！

LIBLINEAR とは，国立台湾大学の研究グループによる
SVM のライブラリである！

……はい．まじめにいきましょう[*1]．

大体上の説明通りではあるのですが，SVM ってなに？という方もいると思います．そもそも SVM とは何かというと，Support Vector Machine という機械学習の手法の一つです．どんなことができるのか，細かいことを考えずに簡単に説明すると，

1. A と B という 2 つの種類に分類できるデータがいっぱいあります．
2. A と B のデータを，いっぱい SVM にあげます．

[*1] LaTeX のフォントでふざけるとなんだか寂しい感じですね……．

3. SVM はもらったデータで頑張って学習します.
4. 学習が終わったら，SVM に A なのか B なのかわからないデータをあげます.
5. SVM は，それが A なのか B なのか予測してくれます.
6. 素敵ですね！[*2]

こんな流れです．つまり SVM は勉強させると分類してくれる素敵な機械なのです．この記事は，SVM についてはこれぐらいの理解で『SVM でいろいろ使って遊んでみよう』という趣旨でお送りします．そのために，SVM のライブラリの一つである LIBLINEAR の使い方から説明して，実際に LIBLINEAR を使っていろいろなものを分類してみます.

想定環境

筆者は OS は Linux, 文字コードは EUC-JP で実験していましたが, 基本的に Windows や Mac の環境でも問題はないと思います．環境に依存する部分にはできる限り複数の環境での説明を入れます.

LIBLINEAR のつかいかた

LIBLINEAR の導入

まずは LIBLINEAR のページ[*3]から zip file か tar.gz をダウンロードします．ページの真ん中あたりからダウンロードできます．このファイルを解凍して出てくるファイルをいくつか使用します．どのファイルをどう使うかは後ほど.

LIBLINEAR のつかいかた

LIBLINEAR には使い方が大雑把にわけて 2 つほどあるのですが，今回は比較的簡単にできる方を紹介します．これは，LIBLINEAR の実行可能形式ファイルに LIBLINEAR が解釈できる形式にしたファイルを渡して学習してもらい，同じように LIBLINEAR が解釈できる形式のファイルで予測してもらうという方法です．基本的にプログラムを書いて LIBLINEAR に解釈できる形のデータを出力するのが主な作業になります．この方法で必要なファイルは，Windows な方は “windows” フォルダの中にある “train.exe” と “predict.exe” です．Linux では make を実行すると生成される “train” と “predict” を使います．ちなみに，C/C++ でコードを書いて LIBLINEAR の関数を直接呼び出して使うこともできます．ここではあまり解説しませんが，こっちの場合，必要なファイルは “linear.h”，“linear.cpp”，“tron.h”，“tron.cpp” と，“blas” ディレクトリに入っている

[*2] 素敵ですよね？

[*3] http://www.csie.ntu.edu.tw/~cjlin/liblinear/

ファイルです．詳しくは LIBLINEAR の README に書いてあります[*4]．

とりあえず試してみよう

分類といっても具体的にはどんな感じなの？と思うかもしれません．なので，まずは見た目にわかりやすく点を分けてみたいと思います．

やってみること詳細

点を分ける，というのは具体的には次のような感じです．

1. 平面を分離する適当な直線を考えます．
2. ランダムに点を打って，直線で分けられた領域によってその点を 2 種類に分類します（A と B とします）．
3. たくさん点を打ったところでそのデータで LIBLINEAR に学習してもらいます．
4. 領域についてはいったん忘れます．
5. またランダムに点を打って，今度は LIBLINEAR さんに分類してもらいます．
6. ここで領域のことを思い出します．
7. ちゃんと領域の通りに分けられてる！すごい！

人間が簡単にできる作業ではありますが，まずはお試しということで簡単な例にしてみました．

学習してもらおう

これから LIBLINEAR に学習をしてもらうわけですが、LIBLINEAR が学習するために必要なデータが二つあります．それは，**ラベル**と**特徴量**です．[*5]ラベルとは，上の説明でいうところの A と B，つまり分類名です．整数値で表すので，今の場合は A は 0，B は 1 とでもしておきましょう．特徴量は，その名の通り特徴をあらわす量です．上の例で言うと，点の特徴をあらわしているのは X 座標と Y 座標ですね．特徴量についてはどんなものを対象にするかで何をとればいいのかが大きく変わってくるのですが，詳しくはそれぞれの章で説明しようと思います．必要なデータがわかったところで，LIBLINEAR に実際に学習してもらう方法に入ります．LIBLINEAR が理解できるデータのフォーマットは次のような感じです．

(ラベル) (特徴量の番号):(特徴量の大きさ) (特徴量の番号):(特徴量の大きさ)

[*4] もちろん英語ですが…….

[*5] 他の言い方もあるようですが今回はこれでいきます．

と特徴量の数だけ続きます．これが 1 つのデータの分で，以下同じ形式でデータの数だけ 1 行に 1 つのデータを書いたものが続きます．それぞれの中身ですが，ラベルは問題ないですね．特徴量の番号，というのはそれぞれの特徴量を 1 から始まる番号に変換したものです.[*6]今回は X 座標を 1，Y 座標を 2 にしましょうか．というわけで例として X=10, Y=5，分類は A のデータがあったとしたら LIBLINEAR にあげるデータは

0 1:10 2:5

となります．これを点の数だけ並べれば学習データは完成です！学習させてみましょう．データのファイル名を TrainData とすると，“./train TrainData”[*7]で実行できます．成功っぽいメッセージは出ましたか？ TrainData.model ができていれば成功で、学習はこれで終わりです！

予測してもらおう

今度は予測なのですが，予測は結構簡単です．予測に必要なデータは特徴量のみです．これをまたさっきと同じデータ形式にするのですが，ラベルも一応必要になります．ラベルがついていれば的中率を出してくれるので，今回は学習データと同じ方法でデータを作ってそれで試すことにしましょう．今度は predict の方を使います．

predict (テストデータ) (モデルデータ) (出力先)

とすれば予測を実行できます．テストデータは今作ったデータで，モデルデータはさっき出てきた.model のファイルです．出力先は適当に output とでもしておけば大丈夫です．実行すると，的中率が表示されて出力先に予測したラベルがずらっと並びます．的中率はどれくらいでしたか？この程度の問題だったら多分ほとんど正解になったと思います．LIBLINEAR はすごいですね！

単語を分けてみよう

さて、試しに点を分けてみたところで、実際に役に立ちそうなものを分けてみることにしましょう。点を分けるのも役に立たないとは言いませんが、もっと人にはできなさそうなことをしてほしいですよね？というわけで、単語分割をしてもらうことにします。[*8]単語分割は、日本語をコンピュータで処理するうえでの基礎となる処理で、そもそも単語ってどういうものなのかとか細かい話が結構いろいろあります。が、今回はある分割済みのデータがあって、その分け方はよくわかんないけど LIBLINEAR さんに頼んで分けられてないデータもその法則に従って分けてもらおう！という感じです。

[*6] **1 から始まる**はわりと間違えやすいので注意.
[*7] Windows ならコマンドプロンプトで “train TrainData”.
[*8] ルールさえわかれば人間の方が正確にできますが…….

特徴はどこ？

ところで，LIBLINEAR さんに予測してもらうためには，ラベルと特徴量のデータをあげなければいけないのでした．今回の場合，文字と文字の間を，分ける or 分けない，で分類してもらえばうまくいきそうです．その場合，分けるか分けないかの判断のもとになりそうなデータは何でしょうか？文字と文字の間を見るので，その前後の文字を見るとうまくいきそうです．具体的にどう見るのかというのはいろいろあるのですが，ここでは単純に前の 1 文字，後ろの 1 文字だけを見ることにします．

LIBLINEAR に理解してもらうために

ラベルと特徴量をどうするかがわかったところで，それらを LIBLINEAR に理解してもらえるように変換しなければいけません．特徴量を前の文字，後ろの文字にすると決まったわけですが，特徴量の大きさはどうするのでしょうか．答えは，出現したら 1，出現しなかったら 0 にする，です．そして，出てきた文字の種類の数[*9]だけ特徴量を持ちます．詳しく説明します．例えば次のような文があったとします．

今日 は 雨 だ 。

この文章のラベルと特徴量をまずは数値に変換しないで列挙すると，

0 前今:1 後日:1 前日:0 後は:0 前は:0 後雨:0 前雨:0 後だ:0 前だ:0 後。:0
1 前今:0 後日:0 前日:1 後は:1 前は:0 後雨:0 前雨:0 後だ:0 前だ:0 後。:0
1 前今:0 後日:0 前日:0 後は:0 前は:1 後雨:1 前雨:0 後だ:0 前だ:0 後。:0
1 前今:0 後日:0 前日:0 後は:0 前は:0 後雨:0 前雨:1 後だ:1 前だ:0 後。:0
1 前今:0 後日:0 前日:0 後は:0 前は:0 後雨:0 前雨:0 後だ:0 前だ:1 後。:1

こうなります．ちょっとごちゃごちゃしてわかりにくいのですが，基本的に前後を分けてそれぞれの文字だけは 1 にして，あとは 0 にするわけです．ちなみにラベルは分割しないを 0，分割するを 1 にしています．これの文字の部分に 1 から順番に数値を割り振っていけばデータは完成です．が，この量だとまだ大丈夫ですが，文が多くなると大変なことになりますね．というわけで，0 の特徴量は書かずに値のあるものだけにして，特徴量の番号順に並べます．今は同じ特徴量は 1 回しか出てきていませんが，文の量が多くなってくると重複してくることになります．というわけで，今度はデータを出力するために文字[*10]を key，番号を value としたハッシュを使うのがよいでしょう．実装の詳細についてここでは詳しくは述べませんが，前述の Web ページに例を置いておきます．また，今回は単語に分割されたデータがないと実験できないのですが，そのデータについても Web

[*9] 前，後ろは分けます．
[*10] 「前今」とか「後は」とか．

ページを参照してください．長くなってしまいましたが，これらを踏まえてできたデータを LIBLINEAR にあげれば学習は完了です！そして同じように作ったデータで予測ができます．ここでひとつ注意なのですが，学習に使ったデータと予測に使うデータでは特徴量の番号が同じものは同じ特徴量に対応していなければいけません．つまり，学習で使ったデータで番号 1 の特徴量が「前今」に対応していたら，予測のデータでも同じようになっていなければいけないということです．

今度は文章！

さあ次は文章です．単語分割はちょっとややこしかったかもしれませんが，文章の分類は単語分割と似ている部分が多いので，単語分割がなんとなくわかれば簡単だと思います．今回はとりあえず Twitter で 2 人の発言を学習して，ある発言が与えられたときにどっちの人なのか予測してもらうことにします．

学習のその前に

Twitter の発言を利用するということで，まずは発言を引っ張ってこないといけないのですが，結構な量が必要なのと，実は単語分割済みのデータが必要になります．Twitter の発言を集めてくるのはスクリプト言語などを使えば結構簡単にできる作業ですし，また単語分割も KyTea[*11]や ChaSen[*12]を使うことで可能です．が，とりあえずここでは簡単に試してもらいたいので Web ページにそのためのデータを置いておこうと思います．

学習しましょう

学習しましょう！いつものようにラベルと特徴量を考えます．ラベルは発言者でいいですね．わかりやすいです．特徴量はどうしましょう．単語分割済みデータが必要ということでピンと来た方もいると思いますが，単語の出現量を使います．ある発言があったとして，その中のすべての単語の出現数が特徴量となります．これも前回と同じように単語を特徴量の番号に直して，かつ値が 0，つまり出現してない単語の特徴量は消します．というわけで，単語単位になることと，前後だけじゃなくて発言の中のすべての単語を取る，というのが単語分割との違いでしょうか．ちなみに，このような特徴量のとり方を Bag-Of-Words と言います．さて，これでデータを作って学習，予測をしてみましょう．うまくいきましたか？うまくいかない場合は前述の Web ページにあるサンプル等も参考にしてみてください．

[*11] http://www.phontron.com/kytea/index-ja.html
[*12] http://chasen-legacy.sourceforge.jp/

そして他の応用へ

文章分類ができるといろいろと応用して遊ぶことができそうです．たとえば自分の好きな話題とそれ以外で分類して自分の好きな話題だけを勝手に抽出できるかもしれません．他にはニュースサイトのカテゴリ別の記事から学習して他の文章もカテゴリに分類できるようになるかもしれません．夢が広がりますね！応用範囲はもっといろいろあると思うので，ぜひいいアイデアを思いついたら試してみてください．

まとめ

結構長くなってしまいましたがこの記事はこれでおしまいです．楽しんでもらえましたでしょうか？この記事では，SVM の細かいところにまったく触れずに機械学習を紹介してきました．適当に遊ぶだけならこれでも十分楽しめると思うのですが，精度を上げようと思うとやはりもっと詳しく知っておく必要があります．興味を持ってもっといいものを作りたいと思った方はぜひ調べてみてください．SVM については，LIBLINEAR を作っている研究チームによる，SVM Guide[*13]を読んでみることをお勧めします．英語ですが，SVM をうまく使うためのコツなどがいろいろ書いてあります．さらに，SVM は機械学習の一分野でしかありません．筆者もあまり機械学習に詳しい方ではないのですが，この記事で機械学習全般にも興味を持っていただけたらうれしいです．

[*13] http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

C golf

eha

はじめに

こんにちは．KMC 2 回生の eha です．ここでは私の趣味の記事を書かせていただくことにしました．

今回の部誌で kirita 君がコンテストの記事を書いているのはもう読みましたか？彼の記事で紹介されているオンラインジャッジシステムの中に **Aizu Online Judge**（通称 AOJ）があります．AOJ は C, C++, Java のいずれかで与えられた問題を解くプログラムを書き，それが正しいかどうかをジャッジします．正しかった場合，それはもう大変ありがたい Accepted をいただけるのですが，この時「ユーザ名」,「実行時間」,「メモリ使用量」と「**コードの長さ**」が記録されます．そして一部の AOJ プレイヤーが「**コードの長さ**」を**どれだけ小さくすることができるか**を競い始めたのです．

コンパイル時に画面いっぱいに警告を出されても，動けば問題ない！

C golf とは？

Code golf では上で少しでてきた「問題を解くプログラムをいかに短く書くか」を競います．使う言語によって短くなるコードは違うので，ここでは AOJ で使用可能な C 言語での golf，つまり **C golf** をとりあげます．

今回の部誌では具体的に 2 問の問題について，実際にどのようにすれば短くなるかを書いていきます．早速見てみましょう．

AOJ 0000 QQ

AOJ に登録しておそらく一番最初に取り組む問題は QQ です．指定された表記で九九を出力するプログラムを書く問題です．普通に解くと下のようになります．

```
#include <stdio.h>
main(){
    int i, j;
    for(i = 1; i <= 9; i++)
        for(j = 1; j <= 9; j++)
            printf("%dx%d=%d\n", i, j, i*j);
    return 0;
}
```

上のコードを基本にし，短くしていきます．まず，ヘッダをインクルードする必要はありません．関数の引数の型チェックをしなくなり，stdin, NULL, EOF などの定数が使えなくなるなどの制約がかかりますが**今は動けばよい**ので消します．余分な空白と改行も**全て**除きます．次に for 文の第 3 式に書かれたインクリメント演算子ですが，これを第 2 式の比較している式の中で実行します．また，計算順序を考えると，比較演算子も短くなります．また，大域変数と main 関数の仮引数は int 型に限り型の省略が可能で，前者は 0 で，後者は AOJ では第 1 引数に限り 1 で初期化されます．

```
i;main(j){
    for(;i++<9;)for(j=0;j++<9;)printf("%dx%d=%d\n",i,j,i*j);
    return 0;
}
```

2 つある for 文を 1 つにまとめます．どうすればよいでしょうか？ j が 9 より大きい時に j に 0 を代入し i をインクリメント，そうでなければ j をインクリメントし，継続条件は i が 9 を超えるまで，とするとうまくいきます．条件分岐は三項演算子を使って書きます．最初に i を 1 で初期化するために main 関数の第 1 引数としています．また，j に 0 を代入するのに論理否定演算子で 0 を作って代入しています．そして return 文を消します（後述）．

```
j;main(i){for(;i<=9;)j++>8?j=!++i:printf("%dx%d=%d\n",i,j,i*j);}
```

これで 64B の QQ のコードができました．地道な作業ですが，思考錯誤して最短解にたどり着いた時はやっぱり嬉しいです．そして**皆さんの中に少しでも興味を持った方がい**

れば，ぜひ体験していただきたいです．さて，実はこの問題の暫定最短解は 62B です．楽しみを奪ってしまうのは申し訳ないので，解答は載せません．ヒントとしては，論理否定演算子を使わずに j にある値を代入することができないかを考えます．比較演算子の結果を使うことはできないでしょうか？ヒントはここまでです．

main 関数の return 0; を消してもいいのか？

AOJ に「**main 関数は必ず 0 を返すようにして下さい．**」と注意書きがあります．何気なく消してしまった `return 0;` ですが，先ほどの注意書きを満たさないと Runtime Error となり，たとえ自分の環境で動くプログラムだとしても Accepted をもらえません．eax レジスタに 0 が記録された状態で終了すれば return 文を書かずに済みます．私はとりあえず `return 0;` を消して提出し，ダメだったらそこで eax レジスタに 0 が入るように工夫（大域変数に 0 を代入する等）するようにしています．

AOJ 2252 koukyoukoukokukikou

図 1　keyboard

これは今年の ICPC の模擬国内予選の A 問題です．キーボードのキーごとに，右手と左手のどちらで打つかを示した図があります．英小文字のみからなる文字列が複数与えられるので各文字列ごとに「タイプするのに手が入れ替わる回数」を求めよ，という問題です．#だけの行を読み込んだ時にプログラムを終了します．例えば “koukyoukoukokukikou” という入力が与えられたなら，全部右手で打つことができるので答えは 0 です．打ちづらいですね．

まず，どの文字をどちらの手で打つかを予めコード中に埋めこむ必要がありますが，26 個分の右 or 左を記録させることができて，短く表せる方法はないか考えてみましょう．真っ先に思い浮かぶのは配列です．右で打つものを 1，左で打つものを 0 として配列に保存しておけば，文字が入力された時に対応する配列を見ればどちらの手で打つのかが分かります．もう少し工夫できないでしょうか？ 2 通りにしか区別する必要がないなら，左=0 と右=1 の並びを 2 進数で表し，それを 10 進数に変換したものを保存します．ABC...XYZ のそれぞれを左を 0、右を 1 で置換したものを 2 進数とみなして，それを 10 進数に変換すると 523298 になります．後はビット演算で入力に対応した部分を 0 or 1 の

形で取り出すことができればよさそうです．これらはビットシフトとビット AND の演算子を用いれば容易に実現できます．演算子の優先順序を考慮して，不要な括弧は外します．

```
int a,s;
main(z,b){
    for(;s=getchar()-35;){
        if(s>0) a+=b!=(z=523298>>87-s&1),b=z;
        else a=!printf(b="%d\n",a-1);
  }
  return 0;
}
```

4 行目で z には 0 か 1 かの数値が入ります．これは入力された文字をどちらの手で打ったかを示します．そして b には直前の文字をどちらの手で打ったかを残しておき，打つ手が変わった時に出力する値を 1 増やします．main 関数の第 2 引数は本来 char* argv[] として使われますが，int 型で宣言することも可能です．この場合，環境にもよりますが，文字列へのポインタ（=0 と 1 以外）で初期化されます．入力文字列の最初の文字に対応する数字が 0 の時と 1 の時で答えが変わってしまうのを避けるため，b は 0 と 1 以外の整数で初期化しておきます．また，“hoge” は hoge という文字列の先頭へのポインタです．これは巨大な整数となります．print 関数で表示後にこれを代入することで前述した問題を避けます．“#” を読み取った時，for 文の継続条件式が 0 となり，プログラムが終了します．26 個の右 or 左のデータを見事 6 つの文字だけで表すことに成功しました（余談ですが，さすがに大会中にこんなコードは書きませんでした）．

ここまできたら，後は QQ 同様にコードを短縮していきます．条件分岐は三項演算子を使い，for 文の空いたスペースに入れられる式があれば入れます．私は 90B の解でひとまず終了していますが，頑張ればもっと短くなるかもしれません．興味のある方は是非挑戦してみてください．

おわりに

参考書籍

『Short Coding ——職人達の技法——』
Ozy（著），やねうらお（監修）
ISBN-10: 4-8399-2523-2
ISBN-13: 978-4-8399-2523-9

この本には C 言語での golf テクニックが実例を交えて丁寧に書かれています．私が最初に買ったプログラミング関連の本でした．読み物としても面白いです．大好きです．

ゴルフ場

Anarchy golf (`http://golf.shinh.org/`)
: 通称「あなごる」です．評価の基準はコードの短さのみ．こちらほどたくさんの言語に対応したゴルフ場は他にないでしょう．自分で問題を出題することができるのも特徴のひとつです．そして猛者揃いです．

MixC++ Codegolf Online Judge (`http://roxion1377.toypark.in/newsys/`)
: ideone を利用した画期的なジャッジシステムです．こちらも評価の基準はコードの短さのみです．1 週間に 1 問出題され，1 週間の期間内にどれだけ短くできるかを競います．期間後は全員のコードを見ることができます．対応している言語は C のみです．

ページ数をあまり割くのも申し訳ない気がしたのでかなり割愛した形になってしまいましたが，私のコラムは以上です．ここまで読んでくださりありがとうございました．

簡潔データ構造

chir

簡潔データ構造とは

コンピュータの性能の発達によってコンピュータで扱うデータの量はものすごい勢いで増加しています．例えば自然言語処理で扱うデータやゲノム情報などは，数 GB のデータとなることも当たり前です．

このような大量のデータはデータの格納領域を節約するために圧縮されることが多いですが，一度圧縮してしまうと復元に時間がかかるため，頻繁にデータを書き換えたり，高速にデータを検索したいような場合にとても時間がかかってしまいます．逆に，高速にデータを検索するためには，索引などをつける必要がありどうしてもデータの格納領域を大きくとる必要があります．

このような大量のデータを圧縮しかつ高速に扱うための簡潔データ構造というものが，最近盛んに研究されています．簡潔データ構造というのは「いくつかの操作を定数時間に行うことができる圧縮されたデータ構造」のことです．bit 列や 2 分木，文字列などに対してこのようなデータ構造がすでに作られています．今回は，この中でももっとも単純なものの一つであり，他の簡潔データ構造の基礎となっている bit 列の簡潔データ構造を実装してみたので，それを紹介してみたいと思います．

bit 列

前の章で簡潔データ構造は「いくつかの操作を定数時間で行うことが……」と書きました．よって簡潔データ構造を考える場合には，どのような操作に焦点をあてるかということが重要になってきます．

bit 列の場合には，ほとんどの場合 rank という操作と select という操作に焦点をあててデータ構造を構築するようです．なので，この 2 つの操作についてどのような操作なのかと，それをいかにして定数時間で実現するのかについて説明したいと思います．以下では，説明のために bit 列の長さを n で表し，i 番目の要素を $bv[i]$ のように書くことにします．また，要素の添字は 0 から $n-1$ までということにします．つまり，最初の要素が

$bv[0]$ であり，最後の要素が $bv[n-1]$ となります．

rank

rank は，$i < n$ である i に対して，0 番目から i 番目までにいくつの 1 もしくは 0 があるか求める操作です．しかし，1 の個数を求める操作を rank_1，0 の個数を求める操作を rank_0 と書くと，$\mathrm{rank}_0(i) = i - \mathrm{rank}_1(i)$ なので，この後は rank_1 だけを考え，ただ rank と書いた場合は rank_1 をさすことにします．例えば，$bv = 1001011010101010$ であったとすると，$\mathrm{rank}_1(7) = 4$, $\mathrm{rank}_0(7) = 3$ になります．

select

select は，i 番目に 0 または 1 が現れる位置を求める操作です．これも rank と同じように select_0，select_1 というように書いて，0 が現れる位置を求める操作なのか，1 が現れる位置を求める操作なのかを区別します．つまり，$\mathrm{select}_p(x) = \min\{k \in [0, n) \mid \mathrm{rank}_p(k) = x\}$ です．

実装方法の比較

自然に持ってみる

まず，ナイーブに bit 列を実装した場合について考えてみます．1 つの bit を記憶するのに，当たり前ですが，1 bit 必要なので全体で n bit の領域が必要です．rank と select に関しては，bit 列の前から順に調べていくしかないため，$O(n)$ の時間がかかります．

前もって計算しておく

次に，多少記憶容量を使って rank と select を高速に求められるような方法を考えてみます．一番単純な方法は，各 bit に対して rank と select を前もって計算しておき，結果を配列に保存しておくというものでしょう．この方法の場合，事前計算に $O(n)$ の時間がかかりますが，一度計算してしまえば $O(1)$ で各操作を行うことが可能です．

記憶領域については rank の結果を一つ保存するのに $\log n$ bit 必要なので，rank の結果をすべて保存するために $n \log n$ bit 程度必要で，select の結果を保存するためには，$m = \#\{i \mid 0 \leq i < n \wedge bv[i] = 1\}$ とするとおよそ $m \log n$ bit 必要になります．また，rank の結果を使えばある bit の値が 0 か 1 かを求められるので，元の bit 列を保存しておく必要はありません．これらをすべて合計すると最悪の場合で，$2n \log n$ bit 必要になります．

もう少し賢い方法

1 つ目の方法より小さい計算量で 2 つの操作が可能でかつ，2 つ目の方法より記憶容量が小さくて済むようなデータ構造があると嬉しいわけですが，実際にそのようなデータ構

造がいくつか考えられており，そのうち記憶容量が $n/\log n + n\log\log n/\log n + n$ bit 程度で済んでしまいかつ rank 操作が $O(1)$，select 操作が $O(\log n)$ で行えるようなデータ構造を今回実装しました．

さらに賢い方法

さらに効率のよい方法として，bit 列中の 1 の個数が m であるような bit 列 B に対して $p = m/n$ とおいて，$H_0(B) = -p\log_2 p - (1-p)\log_2 1 - p$ と定義できる 0 次経験エントロピー $H_0(B)$ を用いて記憶容量が $nH_0(B) + o(n)$ であり，かつ二つの操作の両方が $O(1)$ でできるようなデータ構造も発表されています．

しかし，これはかなり複雑（というか私もあまりわかってないので……）なため今回は紹介しません．

実装の詳細

どのようにして rank 操作を定数時間で実現しているか詳しく説明していきたいと思います．基本的には bit 列をいくつかのブロックに分け，そのブロックごとに予め rank を計算しておくことで実現しています．

図 1

まず，bit 列を長さ $\log^2 n$ の大きさのブロック (SB: Super-Block) に分割します．次に，各 SB を長さ $\log n/2$ のブロック (TB: Tiny-Block) に分割します．そして，各 SB の先頭の rank の結果と各 TB の先頭の rank の結果と直前の SB の先頭との rank の差を保存しておきます．保存した配列をそれぞれ SB，TB と呼ぶことにすると，$\mathrm{rank}(i) = SB[i/\log^2 n] + TB[(i/\log n) \times 2] +$ 残りの $\mathrm{rank}(i)$ という風に書け，残りの部分については，長さが $\log n/2$ 程度なので popcount や表引きなどの方法を用いて定数時間で結果が求められる程度に短くなっています．

select 操作については，$\mathrm{select}_1(j)$ は $\mathrm{rank}_1(k) < j \leq \mathrm{rank}_1(k+1)$ となる k ですから，rank 操作の結果を 2 分探索することで計算できます．なので，select 操作にかかる時間は $O(\log n)$ になっています．

popcount

長さがある程度短い bit 列に対しては，bit 演算の繰り返しで rank を求めることが可能です．以下に，int，つまり 32bit の bit 列にたいする popcount の C 言語で動くコードを示します．

```
int popcount(int x) {
    x = (x & 0x55555555) + (x >> 1 & 0x55555555);
    x = (x & 0x33333333) + (x >> 2 & 0x33333333);
    x = (x & 0x0f0f0f0f) + (x >> 4 & 0x0f0f0f0f);
    x = (x & 0x00ff00ff) + (x >> 8 & 0x00ff00ff);
    return (x & 0x0000ffff) + (x >>16 & 0x0000ffff);
}
```

このコードは nojima さんが公開しているコードをお借りしました．

使ってみる

作成したデータ構造とナイーブに実装した場合とを比べてみました．sb が実装した構造の結果を表しています．表の中の値は，それぞれ rank 操作をランダムな要素に対して 1000 回繰り返すのにかかった時間です．sb の結果には，SB，TB の配列を求める時間も含まれています．また，時間の単位は ms です．

n	100	1000	10000	100000	1000000
ナイーブ	0.007	0.914	84.1059	821.9158	8183.2540
sb	0.0009	0.0100	0.0920	0.5209	5.1729

ナイーブな実装が長さに比例して処理時間がかかっているのに対して，今回実装したデータ構造は非常に高速だということがわかります．

応用

集合

bit 列を用いて，集合を表すことも可能です．0 から $n-1$ までの値を持つ可能性のある集合 S に対して，$i \in S$ ならば $bv[i] = 1$ とした bit 列を作成すれば，$\#\{i \in S \mid i < j\}$ を $\mathrm{rank}(j)$ で求められ，S の中で i 番目に小さい要素が $\mathrm{select}(i)$ で求められます．また，

C++ などの set は平衡 2 分木を用いて実装されていますが，この bit 列での実装を行えば，平衡 2 分木を用いた実装に比べて記憶容量を小さく抑えることが可能です．

まとめ

これからコンピュータの性能がどんどん向上し扱うデータ量はどこまでも増えていくことが予想されます．なので，今回紹介した簡潔データ構造はこれからどんどん重要性を増していくでしょう．実際に文字列についての簡潔データ構造である Wavelet tree などは，文書の全文検索などに使われています．他にもたくさんのデータ構造があるので興味を持った方は，ぜひいろんなデータ構造を調べてみてください．

参考にした資料

今回の記事を書く上で，以下の資料を参考にしました．

- http://codezine.jp/article/detail/260
- http://codezine.jp/article/detail/261
- http://hillbig.cocolog-nifty.com/do/files/SuccinctDataStructure_0607_v2.pdf

Coq を用いた証明駆動開発

@k_hanazuki

はじめに

この記事では OCaml や Haskell などの一般的な関数型言語を使用した経験のあるユーザを対象に，*Coq* を用いたバグのないプログラムの開発手法を説明します．

Coq はフランスの国立情報学自動制御研究所 (INRIA) で開発された対話型の証明支援系で，数学的な定義や定理を形式言語で記述することができ，それらの定理に対してユーザが与えた証明の正しさを機械的に検証することのできるシステムです．また，OCaml ライクな文法を持つ純粋関数型言語でもあり，ユーザが定義したプログラムの性質についても証明を行うことができます．

このような *Coq* の機能をプログラム開発の分野に応用すると，アルゴリズムを実装する前にそのアルゴリズムが満たすべき性質を数学的な定理として記述し，実装したアルゴリズムに対してそれらの定理を証明し検証することで実装が正しいことを確認するという，「証明駆動開発」と呼べる開発手法をとることができます．これはテスト駆動開発において，実装の前にテストを記述し，実装後に実装の正しさをテストによって評価するのと対応しています．

以下では簡単な例としてソート関数を実装したいという場合を想定して，証明駆動開発について解説します．

プログラムの満たすべき性質を定義する

関数を定義する前に，まずソートのアルゴリズムが満たすべき性質を考えてみましょう．

いま，ソートの対象となる値の型を T とし，型 T の任意の 2 つの値は比較可能である——全順序が定義されている——と仮定します．型 T の上でのソート関数は入力として型 T の値のリスト $\boldsymbol{l}$ を受け取り，出力として $\boldsymbol{l}$ をソートした結果のリスト（$\boldsymbol{l}$ のソート列）を返します．リスト $\boldsymbol{l}$ のソート列とは，それぞれ以下に定義するように，$\boldsymbol{l}$ の並べ替えになっていてかつ順序にしたがって並んでいるようなものと決めます．

定義. リスト $\boldsymbol{l}'$ がリスト $\boldsymbol{l}$ の並べ替えであるとは，リスト中の 2 要素の位置を入れ替える操作を $\boldsymbol{l}$ に対して 0 回以上行って $\boldsymbol{l}'$ が得られることとする． ■

定義. リスト $\boldsymbol{l} = a_0 a_1 \cdots a_{n-1}$ が順序 $\leq$ に従って並んでいるとは，$a_0 \leq a_1 \leq \cdots \leq a_{n-1}$ が成り立っていることとする． ■

それではこれらの諸性質を *Coq* で表現してみましょう．まず，*Coq* の標準ライブラリには全順序の定義された構造を表すモジュール型[*1]Orders.TotalLeBool' が存在するのでこれを使います．今回つくる Sort モジュールは汎用性を考えてこの TotalLeBool' 型のモジュールを引数として取る関手[*2]として定義します．TotalLeBool' 型のモジュールはメンバとして型 t と，t −> t −> bool 型の関数 leb を持っています．この t 型を用いてソート関数 sort の型は list t −> list t とかけるので，いまはひとまず仮定としてこの型の関数が存在することだけを宣言しておきましょう．リストが別のリストの並べ替えになっているかどうかと，ソートされているかどうかを表す命題はそれぞれ Sorted.LocallySorted と Permutation.Permutation という名前でライブラリに定義されているのでそのまま使うことにします．

ここまでのことを *Coq* の言葉で書くと次のようになります．

```
Module Sort (Import X: TotalLeBool').
Local Coercion is_true: bool >−> Sortclass.
Local Notation Sorted := (LocallySorted leb) (only parsing).

Hypothesis sort: list t −> list t.

Theorem sort_permuting xs: Permutation xs (sort xs).
Proof.
Admitted.

Theorem sort_sorting xs: Sorted (sort xs).
Proof.
Admitted.

End Quicksort.
```

ここから，関数 sort を定義して（**Hypothesis** を消して実際の定義を書きます），2 つの定理 sort_permuting と sort_sorting の証明を行い（**Admitted** を消して実際の証明を書きます），コンパイルが通れば終了です．

[*1] モジュールに対するインターフェイス定義のようなものです．
[*2] 引数と戻り値がモジュールである関数のようなものです．

プログラムを実装する

さて，それではソート関数 sort を定義してみましょう．ソートアルゴリズムとしてクイックソートを使うことにします．

```
Definition filter_gt p := filter (fun x => x <=? p).
Definition filter_le p := filter (fun x => negb (x <=? p)).

Function sort (xs: list t) {measure length xs}: list t :=
  match xs with
  | nil => nil
  | p :: xs => sort (filter_gt p xs) ++ p :: sort (filter_le p xs)
  end.
```

関数 sort は，リストの先頭をピボットとして分割をおこない，前半と後半に対して再帰的にソートを行うという，関数型言語の入門書に出てきそうなよくあるクイックソートの定義にしました．一般にアルゴリズムの複雑さに応じてその関数の性質を証明するのは面倒になるので，今回はこのくらい簡単な定義にしておきましょう．

さて，普通の関数型言語ではこれで sort 関数の定義はできているのですが，*Coq* の場合はそうはいきません．*Coq* では再帰する関数を定義しようとするとき，その関数が必ず停止することを示さなければならないのです．停止性が自明な場合は処理系に任せることもできますが，sort の場合は残念ながらうまくいきません．停止性を証明するためには，再帰呼び出しに与える引数のサイズが呼び出しの度に小さくなっていくことを言えれば十分です．なぜなら，リストが無限に小さくなりつづけるということはありえないからです．そこで，引数のサイズを測る尺度を **measure** length xs のように与えています．

ではさっそく sort の停止性の証明をしてみましょう．*Coq* の処理系に上のコードを入力すると次のような「証明モード」に突入します．

```
2 subgoals
______________________________________(1/2)
forall (xs : list t) (p : t) (xs0 : list t),
xs = p :: xs0 -> length (filter_le p xs0) < length (p :: xs0)

______________________________________(2/2)
forall (xs : list t) (p : t) (xs0 : list t),
xs = p :: xs0 -> length (filter_gt p xs0) < length (p :: xs0)
```

これは，今から証明しなければならない命題——ゴールとよびます——が 2 つ存在することを意味しています．それぞれ，横線のすぐ下に書かれているのが示すべき帰結で，横線の上には——今はなにもありませんが——帰結を示すために使ってもよい仮定が表示されます．

Coq での証明はゴールに対してタクティクスと呼ばれるコマンドを適用することで進めていきます．この場合は，命題の前提を仮定に引き上げる intros というタクティクを使うことで次のようなゴールに変形できます（これは自然演繹での「仮定を落とす」操作の逆に相当します．あるいは演繹定理と呼ばれています）．

```
p : t
xs : list t
------------------------------------(1/2)
length (filter_le p xs) < length (p :: xs)
```

リストは帰納的に定義された型なので xs の構造に関する帰納法で証明できそうです．induction xs というタクティクを使ってみましょう．

```
p : t
------------------------------------(1/3)
length (filter_le p nil) < length (p :: nil)

------------------------------------(2/3)
length (filter_le p (a :: xs)) < length (p :: a :: xs)
```

ゴールが複数に分裂しました．1 つ目が帰納法のベースケース——nil の場合の証明——で，2 つ目はインダクションステップ——cons の場合の証明——に相当します．2 つ目以降のゴールには仮定が表示されないので気をつけてください．

ベースケースの帰結を頭の中で計算してみると不等号の左辺は 0 で右辺は 1 ですので，不等式は明らかに成り立っています．このような簡単なゴールの場合は auto というタクティクを使うと処理系が自動で証明をすすめてくれます．

```
p : t
a : t
xs : list t
IHxs : length (filter_le p xs) < length (p :: xs)
------------------------------------(1/2)
length (filter_le p (a :: xs)) < length (p :: a :: xs)
```

ベースケースの証明がおわり，インダクションステップの証明に移りました．関数適用を評価して式を簡単にする simpl というタクティクを使ってみます．

```
p : t
a : t
xs : list t
IHxs : length (filter_le p xs) < length (p :: xs)
------------------------------------(1/2)
length (if negb (a <=? p) then a :: filter_le p xs else filter_le p xs) <
  S (S (length xs))
```

式は長くなっていますが，より具体的な値に変形されています．S という関数[*3]がでてきましたが，これはペアノの公理での後者関数，すなわち自然数に 1 を加える関数です．

帰結の中に **if** 式が出てきたので場合分けで証明を進めてみます．a <=? p の値で場合分けしたいので，case leb というタクティクを使います[*4]．場合分けを行うと少しややこしい式が出てきますが，更に simpl を実行すると次のようになります．

```
p : t
a : t
xs : list t
IHxs : length (filter_le p xs) < S (length xs)
______________________________________(1/3)
length (filter_le p xs) < S (S (length xs))

______________________________________(2/3)
S (length (filter_le p xs)) < S (S (length xs))
```

どちらのゴールも帰納法の仮定を使えばすぐに示せそうです．実際，算術の知識を使用した自動証明 auto **with** arith で証明が完了します．もし手動で証明を行いたければ，不等号<の推移性を使えばよいでしょう．そのための transitivity というタクティクが存在します．

以上をまとめると sort の停止性の証明は次のようになります．

```
Proof.
intros _ p xs _.
induction xs.
  auto.
  simpl; case leb; auto with arith.

intros _ p xs _.
induction xs.
  auto.
  simpl; case leb; auto with arith.
Defined.
```

複数のタクティクをセミコロンで結んでいるのは，それらを前から順に実行することを意味しています．ただし，セミコロンの前のタクティクでゴールが分裂した場合，セミコロンの後のタクティクは分裂後のゴール全てに適用されます．場合分けで分裂したゴールをどちらも auto **with** arith で証明したことを思い出してください．

[*3] 厳密にはこれは関数ではなくて，nat 型（自然数型）のコンストラクタです．
[*4] leb は演算子 <=? の本名です．

プログラムの満たすべき性質を証明する

目的の関数が実装できたので，これが先に定義した定理を満足する定義になっていることを確認しましょう．細かい証明の技法を解説していては記事が長くなりすぎるので，ここでは簡単に証明の方針の一例を説明します．

sort_permuting

sort_permuting を示すにあたって，まず補題としてリスト xs のパーティションは xs の並べ替えになっていることを示します．この証明では任意の x について，x <=? p または negb (x <=? p) のどちらか一方が成り立つことを使います．

```
Lemma Permutation_partition xs p:
  Permutation xs ((filter_gt p xs) ++ (filter_le p xs)).
```

この補題を用いると p :: xs が (filter_gt p xs) ++ p :: (filter_le p xs) の並べ替えであることはライブラリで提供されている並べ替えに関する定理を使って示せます．

sort_sorting

sort_sorting の証明では補題として，パーティション列の前半と後半がともにソート列であるとき，全体としてもソート列であるということを示します（Forall は第 2 引数に与えられたリストの全ての要素に対して第 1 引数の命題が成り立っていることを表す命題です）．この証明で leb が全順序であることを使います．

```
Lemma Sorted_partition p xs ys:
  Sorted xs -> Sorted ys ->
  Forall (fun x => (x <=? p) = true) xs ->
  Forall (fun x => (x <=? p) = false) ys ->
  Sorted (xs ++ p :: ys).
```

この補題をうまく使えるように以下の 2 つのほぼ自明な命題を示すことができれば，後は若干の変形のみで sort xs がソート列であることを証明できます．

```
Proposition filter_gt_le p xs:
  Forall (fun x => (x <=? p) = true) (filter_gt p xs).
Proposition filter_le_gt p xs:
  Forall (fun x => (x <=? p) = false) (filter_le p xs).
```

プログラムを使用する

これで sort が正しく定義されていることが確認できました．このバグのないソート関数を他のプログラミング言語から使えるようにしてみましょう．

まず自然数上でのソート関数を作るために，自然数に全順序を入れた構造を作ります．ソート対象となる値の型と順序関数を定義して，全順序になっていることを示す必要があります．

```
Module Nat <: TotalLeBool.
  Require Arith.Compare_dec.
  Definition t := nat.
  Definition leb := Compare_dec.leb.
  Definition leb_total x y: leb x y = true \/ leb y x = true.
  Proof.
  elim (Compare_dec.le_ge_dec x y); [left | right];
  auto using Compare_dec.leb_correct.
  Defined.
End Nat.
```

Sort はモジュールを引数にとってモジュールを返す関手でしたから，Nat モジュールを渡すと自然数の上でのソート関数を含むモジュールが出来上がります．

```
Module NatSort := Sort Nat.
```

そして，自然数のソート関数を OCaml と Haskell のコードとして抽出してみます．

```
Extraction Language Ocaml.
Extraction "sort.ml" NatSort.sort.

Extraction Language Haskell.
Extraction "sort.hs" NatSort.sort.
```

すると，それぞれ以下のようなコードが得られるはずです（メインの部分だけを抜粋しています）．

```
(* sort.ml -- OCaml 版のクイックソート *)
let rec sort_terminate = function
  | Nil -> Nil
  | Cons (p, xs0) ->
    app (sort_terminate (filter_le p xs0))
        (Cons (p, (sort_terminate (filter_gt p xs0))))
```

```
-- sort.hs -- Haskell 版のクイックソート
sort_terminate :: (List Nat) -> (List Nat)
sort_terminate =
  and_rect (\_ _ xs ->
    let {
     hrec xs0 =
       case xs0 of {
        Nil -> Nil;
        Cons p xs1 ->
         sig_rect (\rec_res _ ->
           sig_rect (\rec_res0 _ -> app rec_res (Cons p rec_res0))
             (hrec (filter_gt p xs1))) (hrec (filter_le p xs1))}}
    in hrec xs)
```

これらのコードの正しさは証明によって保証されています！

おわりに

この記事では，*Coq* を用いて正しさの厳密に証明されたプログラムを書く方法を取り上げました．しかし気をつけなければならないのは，定理が導出可能であることは処理系によって保証されますが，定理の定義自体の正しさを保証するのは依然プログラマの責任として残るということです．すなわち，プログラムの「正しさ」を定義するのも人間の仕事なので，ここにバグの入り込む余地があるのです．一般のアルゴリズムに対して，そのアルゴリズムが正しいことの必要十分な条件を探してくるのは容易ではありませんし，その条件が正しいことはまた別の方法で証明しなければなりません．また，そのような条件を見つけられたとしても *Coq* の表現に落とす段階でミスをする可能性が常につきまといます．このようなミスを減らすには，同じ条件を複数の同値な方法で定義して，*Coq* の上でそれらが同値であることを証明してみるといった方法が有効かもしれません．

今回は *Coq* をプログラムの検証器としての側面から紹介しましたが，この手の形式証明は数学的なパズルとしても大変おもしろいものです．興味を持たれた方は是非遊んでみてください．

この記事で取り上げたプログラムと証明のコンパイル可能なソースコードはブログ (`http://d.hatena.ne.jp/k_hanazuki`) に掲載しています．

とびうおのリア充物語（ユニクロ編）

tobiuo

この部誌はコミケで配られることを想定して書いているので，きっと読者の皆様の中にはとても文明人のものとは思えない服を着ていらっしゃるかたもいらっしゃるでしょう．安心してください，僕も半年前はそうでした．僕のファッションセンスは，今の KMC 代表に「まるで下着で街を出歩いているみたいだ」と言われるようなものでした．彼の指摘は半分間違っていて，実際には着る服が 1 着もなくなって，パーカーの下に本当に下着を着て学校に登校していました．

僕のような深刻な布不足に陥る原因ははっきりしていて，それは自分で服屋にいかないことです．読者の皆様の中にも，この世に生をうけ産湯につけてもらってから今まで服は母親が買ってきたバーゲン品という方はいらっしゃるでしょう．この記事ではそういう人たちのために，僕が野蛮人の状態から進化して正常な人間の思考になり，恐れていたユニクロに入店できるようになるまで成功秘話を書こうと思います．

決意

小学校の頃「一緒に京大にいこうねー」と互いに無邪気に笑い合っていた友達と，大学生になって同窓会で会うと切ない思いをすることが度々あります．けれども，僕の友達みたいに「受験時代に付き合っていた彼女は，付き合っている間にミス神戸になった」などとしゃべれば，俺が受験に苦しんでいた時期に遊んでいた罰だざまあみろという気分になるものです．同じ男子校に通っていながらここまで華やかさが違っていたんだと気づいた 2 回生の冬，僕はそろそろ遅れをとりもどすべきなんだと自覚したものでした．

時は流れて成人式の日，親が格好付けて買ってきた Brooks Brothers のジャケット（5 万円）を俺いかしてるぜと意気込んで着ていき「ビバ 70 年代」「君は後ろから見ると教師のように見える」と言われた飲み会の後，僕はファッション雑誌を買いました．最新のファッションにつつまれた細身の男の子たちをみて，服屋に入るためにはまず痩せなければいけないと，僕はこの段階で気づきました．

ダイエット

ダイエットの仕方なんて今まで興味がなかった僕は，KMC の皆さんから何をすればいいか教えてもらいました．そして次の 2 つをこなせばいいことがわかりました．

1. 普通のデジタル体重計に USB 通信機能をつける
2. 京都から比叡山を越えて滋賀県に行く道（山中越）を自転車で走破する

1 つめは「USB 体重計に乗った瞬間に Twitter に自分の体重がポストされるようになれば，体重に対する意識が高まり痩せるだろうという」という期待によるものです．はじめて触る AVR に苦戦して，結局まだできてません．やる気のある電気電子工学の人にやらせないといけません（僕は農学部です）．

2 つめはただの運動です．1 月なので外界の気温が 10 度，山頂付近の温度が-2 度といった環境は厳しいですが，美しい森の風景や琵琶湖を眼下に収めながらママチャリで走る（押しながら歩く）のはよいものです．帰りのついでに大文字山にのぼって湧き水を汲んで，1 日室温で置いてから飲み，胃腸を 1 週間やりました．

これらの文武両道なダイエットのおかげで，僕は 1 ヶ月で 7 kg の減量に成功しました．

いざユニクロへ

ダイエットのおかげで，僕とユニクロの間に大きく横たわっていた溝が埋められました．僕が居候している先の祖母は「せっかく家から 100 m の位置にユニクロがあるのに，そこに入るのに本当に 2 年かかってしまったな，さあ行ってきなさい」と，僕に 5000 円を握らせながら感慨深そうでした．

その後ユニクロの目の前に行って入るのを諦めるのを 2 回繰り返した後，僕はついにユニクロに入ることができました．今着ているズボンのほとんどはユニクロのジーンズです．

まとめ

あなたがかつての僕と同じようにユニクロに対して恐怖感をもってらっしゃるならば，僕と同じ手順を踏み，ユニクロと自分との距離感を近づけてもらえばいいと思います．

OBのページ

2011 年 ACM ICPC 国内予選を終わって

2011 年 ACM ICPC 国内予選を終わって

鴨 浩靖 wd@ics.nara-wu.ac.jp

ACM 国際大学対抗プログラミングコンテスト (ACM ICPC) は，大学生を対象とする世界最大規模のプログラミングコンテストです．1976 年に始まりました．日本の大学のチームが初めて参加したのが，1997 年です．翌年 1998 年からは日本でもアジア地区予選が開催され，日本からも本格的に参加するようになりました．

ACM ICPC には，同じ大学の学生（大学院生も可．ただし，大学入学から 5 年以内）3 名でチームを組んで参加します．

日本から出場する場合，まず，国内予選に参加します．国内予選はオンラインで実施されます．選手は原則として所属大学等からインターネット越しに問題を入手し，インターネット越しに解答を提出します．制限時間内に解けた問題の個数を競います．解けた問題数が同じ場合は解くまでの時間が短いほうが上位となります．基本的には上位から順に予選通過チームが選ばれますが，同じ大学等のチームが多数通過しないための補正ルールがあります．そのため，上位の成績をあげながらより上位に同じ大学等のチームがいるために予選通過を逃すチームもあります．

国内予選通過チームはアジア地区予選に進みます．アジア地区予選は，実際に会場に集まって大会側の用意した機器を使って競技します．アジア地区予選はアジアの複数の都市で日程をずらして開催されますが，日本の大学のチームは，通常は，日本で開催される大会に出場します．ただし，1 チームはアジア地区予選の 2 大会に出場できるルールですので，日本国外のもう一カ所にも出場するチームもあります．アジア地区予選を通過できるのは，1 大学につき 1 チームだけと決まっています．そのため，同一大学の異なるチームがアジア地区予選の異なる大会でそれぞれ上位の成績をあげても，両方が世界大会に進むことはできません．

今年度（2011 年度）の国内予選は 6 月 24 日に行われました．11 月 12–14 日にアジア地区予選福岡大会が，九州大学伊都キャンパスを会場に開催されます．世界大会は，来年の 3 月か 4 月の見込みです．

筆者は，2011 年度アジア地区予選福岡大会の審判の一人として，国内予選の作題および判定を務めました．このあと，福岡大会の作題および判定も務める予定です．このたびの国内予選を審判側から見たところを，問題ない範囲で紹介します．ただし，本稿はあくまでも個人的見解であり，審判団を代表するものではありません．

操作ミスについて

今年度は，選手が操作ミスで解答の提出に失敗するのを多数見ました．まだ，きちんと数えていませんが，昨年度と比べても操作ミスが増えている印象があります．操作ミスで実力が発揮されないのはくやしいでしょうから，来年度以降の出場を考えている方は，操作ミスのないよう，傾向と対策を知っておいてください．

事例　1 回の提出で終わったものと勘違いする．

国内予選は，オンラインでそれぞれの所属大学等の設備を使って競技する都合で，特殊な提出方法になっています．問題とともに判定用入力データその 1 が公開されています．チームは，作成したプログラムとともに，判定用入力データその 1 を入力して実行した結果得られた出力も提出します．提出した出力結果が合っていれば，判定用入力データその 2 がシステムから送られてきます．選手は，同じプログラムに判定用入力データその 2 を入力して得られた結果を，プログラムとともに提出します．この，2 回目が必要なことに気づかず，正解のはずなのに結果が反映されないと思っていたチームがありました．

対策

- ルールをよく読みましょう．
- 試合は夕方から行われますが，当時の昼に，システムに慣れるためのリハーサルが行われます．リハーサルには必ず参加して，解答の提出方法を把握しましょう．

事例　2 回目の提出でプログラムを書き換えてしまう．

2 回目の提出では，1 回目と同じプログラムを提出しなくてはなりません．ところが，1 回目と 2 回目でプログラムが違ってその問題を通過できないチームが，毎年，続出します．調べてみると，そのほとんどは，入力ファイル名か出力ファイル名をプログラム中にハードコードしているのが原因でした．ファイル名をハードコードしているため，2 回目の実行の際にプログラム中のファイル名の部分を書き換えてしまい，その結果，同一プログラムでないと判定されたのです．

対策

- ルールをよく読みましょう．
- リダイレクションを活用しましょう．そうすれば，プログラムにファイル名をハードコードする必要がなく，プログラムを書き換えてしまう事故を防ぐことができます．

事例　提出された出力結果に，余分な行が含まれている．

提出された出力結果に余分な行が含まれていて，それを除けば正解なのに誤答と判定されるのも続出しました．あとで調べたところ，そのうちの多くはプログラムが間違っているのですが，プログラムを審判側で実行すると正しい結果が出力されるのに，提出された実行結果には余分な行が含まれている例もありました．今のところ推測でしかありませんが，出力結果をカットアンドペーストするさいに操作ミスをした可能性が高いです．

対策
- リダイレクションを活用しましょう．
- カットアンドペーストではなく，ファイルのロードの機能を使いましょう．

事例　実行ファイルを提出してしまう．

提出しなくてはならないプログラムというのはソースファイルのことですが，誤って実行ファイルを提出したチームも多数ありました．

対策　これは，審判側で対策をとりました．実行ファイルを提出すると審判システムが検出し，自動的に警告を返します．あらためて，提出し直してください．

問題について

今年度の国内予選では，7 問出題されました．制限時間は 3 時間でした．問題は，ACM 日本支部のページ `http://www.acm-japan.org/` からたぐっていけば入手できます．

今回（2011 年度）の国内予選には過去最高の 311 チームのエントリがありました．今回の戦績は以下の通りです．

正解数	7	6	5	4	3	2	1	0
チーム数	0	4	11	22	39	110	80	45

特別枠を除く最下位予選通過チームは，4 問解いています．

また，問題ごとの正解チーム数は，以下の通りです．

問題	A	B	C	D	E	F	G
正解チーム数	266	184	77	37	16	0	4

正解チーム数だけから判断すると，最後の 2 問を除いて，易しい順に並んでいると見ることができるでしょう．

誌面の都合で，最初の 4 問だけ簡単に紹介します．

■問題 A チェビシェフの定理

問題概要　正整数 n に対して，n より大きく $2n$ 以下をみたす素数 p の個数を求めよ．

想定解法 1　（推奨）エラトステネスの篩で素数判定表を作り，範囲内の素数を数える．

想定解法 2　（遅い）範囲内の整数に対して，試し割り法で素数判定して，素数を数える．

■問題 B 世界の天秤

問題概要　文字列が丸括弧 “(”, “)” と角括弧 “[”, “]” についてバランスしているかどうか判定せよ.

想定解法 1　（推奨）スタックを使用する．文字列を左端から走査し，左括弧が来たらスタックに積み，右括弧が来たらスタックから降ろして括弧の対応を判定する.

想定解法 2　（遅い）まず，括弧以外の文字をすべて削除する．次に，対応する括弧を内側から順に削除する．すべて削除できたら OK.

■問題 C 同色パネル結合

問題概要　正方形のパネルが長方形に並んでいる．パネルは 6 色のうちのどれかである．同色のパネルが辺で接しているとくっつく．左上角のパネルは色を変えることができ，その時，くっついているパネルも同じ色に変わる．5 回色を変え，最終的に指定の色にし，左上角にくっついているパネルの枚数を最大にせよ.

想定解法 1　全探索．素朴に考えると，6^5 個 ($= 7776$) の探索が必要なようだが,

- 最後の塗り替えは指定色に決まり.
- それ以外の塗り替えでは現在の左上角の色に塗り替えても無意味.

を考慮すると，$(6-1)^4 \times 1$ 個 ($= 625$) の全探索ですむ.

■問題 D そして，いくつになった

問題概要　テーブルに何枚かの円盤が置かれている．別の円盤が上に重なっていない円盤の中に同じ色のものが 2 枚あれば，それらを同時に取り除くことができる．これを繰り返すことで，最大，何枚の円盤を取り除くことができるか？

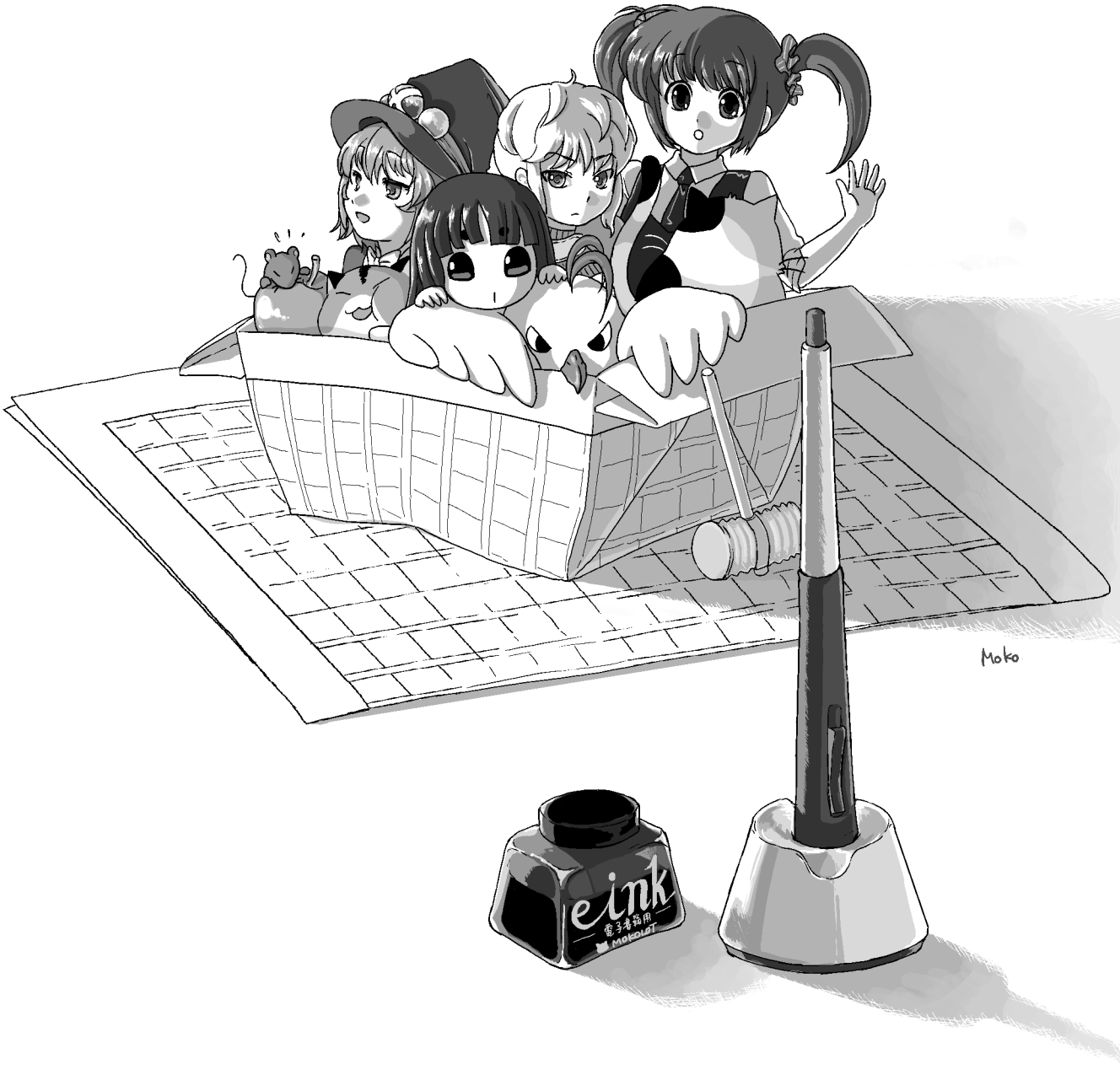
なお，色は 4 色．1 色あたりの円盤の枚数は 6 枚以下.

想定解法 1　メモ化による枝刈りつき深さ優先探索.

想定解法 2　状態の合流を配慮した枝刈りつき幅優先探索.

あとがき

編集統括 seikichi



あとがき

seikichi

編集長の seikichi です．この度は独習 KMC vol.1 を読んでいただき，ありがとうございました．

前回の部誌は KMC というサークルを紹介する内容が中心でした．今回は技術的な内容を増やそう，という思想のもとで記事が書かれました．楽しんで頂けたなら幸いです．

ご意見・ご感想があれば Twitter のハッシュタグ `#kmc_bushi` をご利用下さい．

独習 KMC vol.1

2011 年 8 月 13 日　初版発行

著作・発行	京大マイコンクラブ
編集統括	seikichi
編集補佐	moko, hidesys
表紙イラスト	madaragi
中表紙イラスト	順に tj1990, moko, gire, crys, moko
校正・監修	All KMC members. **Thank you very much!**
メールアドレス	`info@kmc.gr.jp`
Web	`http://www.kmc.gr.jp`

落丁・乱丁の際は在庫がある限りお取り替えいたします．上記のメールアドレスまでご連絡ください．

独習KMC vol.1

■利用分類

部誌

■製品分類

KMC

■レベル

初級　中級　上級

Kyoto University Micro Computer Club