CSのためのACM/ICPC対策マニュアル

井上祐馬、畑中広大、室田美帆、渡邊僚

平成23年1月21日

はじめに

このマニュアルは、大学生が参加するプログラミングコンテスト ”ACM-ICPC ”において国内予選の突破を支援するためのマニュアルである。

世間にはこの国内予選を含む様々なプログラミングコンテストの対策を行っている書籍やサイトが存在するが、これらの情報は一般化されすぎているために、

- 国内予選突破のみを考えると、レベルが高すぎる内容も含んでいる
- 内容が網羅的すぎて、勉強するための優先順位がない
- CS の学生がもう知っている知識についても触れているので、その部分が無駄

などのデメリットも存在する。

もちろんこれらの情報を頭に入れておくことは悪いことではないが、無駄を省き、重要と思われる内容を優先的に勉強することで効率化を図れる。他の知識や高度な考え方などは、必要なものを習得してから身につければ良い。

本マニュアルでは、このような考えに基づいて、”国内予選突破 ”に必要な知識の習得に主眼をおいた対策を行う。

まず 1 章において ACM-ICPC の概要について簡単に説明し、2 章で事前に知っておきたい一般の注意事項を述べる。3 章ではバグの防止、及び対処に関する内容をまとめている。4 章で過去の国内予選のデータを分析し、傾向を掴むことで、優先的に学習すべき知識が何であるかを特定する。4 章の分析結果から、アルゴリズム・入出力への対策の必要性があるため、5 章ではその傾向分析に基づいた対策を、例題を交えて解説し、6 章で入出力やデータ構造などのテンプレートを記載している。

このマニュアルを利用し、一人でも多くの CS 学生が国内予選を突破することを願う。

目次

1 ACM/ICPCについて **4**
1.1 ACM/ICPCとは . . . 4
1.2 プロコンのルール . . . 4
1.3 練習環境 . . . 5

2 注意事項 **5**
2.1 事前準備 . . . 5
2.2 コーディング前 . . . 6
2.3 コーディング時 . . . 6

3 バグについて **7**
3.1 バグを作らないために . . . 7
3.2 バグの潰し方 . . . 8
3.2.1 ループに関するバグ . . . 8
3.2.2 変数に関するバグ . . . 9
3.2.3 アクセスに関するバグ . . . 9
3.2.4 その他のバグ . . . 9

4 傾向分析 **10**
4.1 突破最低ライン . . . 10
4.2 アルゴリズムの出題傾向 . . . 11
4.3 データ構造の頻出傾向 . . . 12
4.4 毎年の出題傾向 . . . 13

5 対策と例題 **14**
5.1 シミュレーション . . . 14
5.1.1 Hanafuda Shuffle (ACM-ICPC Japan Domestic, 2004-07-02) . . . 14
5.1.2 Next Mayor (ACM-ICPC Japan Domestic,2009-) . . . 17
5.2 探索 . . . 19
5.2.1 Get Many Persimmon Trees . . . 19
5.2.2 Curling 2.0 . . . 22
5.3 数論 . . . 27
5.3.1 Dirichlet's Theorem on Arithmetic Progressions . . . 27
5.4 動的計画法 . . . 30
5.4.1 Problem B Balloon Collecting(ACM-ICPC Asia Regional Contest in Tokyo, 2010-12-12) . . . 30

6 ライブラリ **35**
6.1 入力 . . . 35
6.1.1 通常の入力 . . . 35
6.1.2 マップ構造 . . . 36
6.1.3 文字列の入力 . . . 38

6.2 単方向リスト . . . 38
6.3 スタック . . . 42
6.4 キュー . . . 44
6.5 素数 . . . 46

7 参考文献・Web ページ 48

1 ACM/ICPCについて

1.1 ACM/ICPCとは

ACM/ICPCとは、ACM(the Association for Computing Machinery)という世界最大の教育及び科学コンピューティング協会が主催する、大学生を対象とした国際的なプログラミングコンテストのことである。このコンテストでは、1チームは3名の学生からなり、できるだけ多くの問題をできるだけ短時間で解くことを競う。以下ではこの大会をプロコンと略す。

日本からの参加チームは、まずインターネット上で行われる国内予選に参加し、成績上位のチームがアジア地区大会に進出する。さらにアジア地区大会における成績上位のチームは、世界大会に出場する。大学生のプログラミング技術を競う世界大会はこの大会が唯一であり、世界大会に出場する大学やチームには国内的・国際的にも高い評価が与えられている。

CSコースからも毎年出場者がいるが、このマニュアル作成当時には国内予選突破はほとんどなかった。しかし今後、このマニュアルを使って国内予選突破が恒常的になることを願っている。

1.2 プロコンのルール

- 競技ルール
  - 競技参加者が使用できる言語は **C,C++,Javaのいずれかのみ**である。
  - 競技に使用するマシンにサンプルプログラムなどのを準備してはいけない。また、CD-Rなどの外部メディアの使用、電卓、電子辞書、携帯電話の使用も禁止される。ただし、**印刷物の持込は許可される。**
  - 参加者が競技実施中に参照してよい電子的な情報は, 問題の閲覧や解答の提出を行うWebページと、競技に使用するマシン上に存在するマニュアル類(manなど)に限る。そのため、問題の閲覧・解答の提出を行うWebページ以外のページにアクセスすることは禁止される。
  - 競技中、参加者は自チーム以外の者(補欠、コーチ、監督者、他チームのメンバーなど)と相談してはいけない。そのため、審判団以外とのチャット・電子メールの送受信も禁止される。
  - **計算機は1チームにつき1台のみ使用できる。**また、使用する計算機にはマウスとキーボード1対以外の入力装置は使用できない。ただし、プリンタの使用は許される。

- 成績判定ルール
  - 開始時刻になると事前に通知されたwebサイト上に問題が公開され、各チームは好きな問題を選んでプログラムを書き、与えられた入力データに対して手元のマシンで計算する。そして出力(実行結果)とソースプログラムをWeb経由で提出し、判定を受ける。
  - 「同一のプログラムが、まだ解いていない入力データ2つに対し連続で正解を出力することができた」ことをもって、その問題を解くことができたとみなす。入力データは各問大ごとに4つ用意されており、最初のデータは問題文中に示される。一つの入力データについて正解を出力できた場合、次の入力データが渡され、それに対し同じプログラムで正解を出力できるかどうか、を判定する。

– 順位は、正解数の多いチームが上位となり、正解数が同じ場合は解くことができた各問題について、下記により算出された**ペナルティの和が小さい方が上位**となる。

  * 競技開始時間からその問題が解けるまでに要した時間
  * 解けた問題の、不正解の提出回数 1 回につき 20 分

以上の通り、データという形での持込はできないが、印刷物であれば持込は許される。このマニュアルを持ち込んで国内予選に臨むことを想定して、このマニュアルは製作されている。

また、解けた問題数が同じでも解いたのが遅ければ国内予選を突破することはできない。いかに早く問題を解くか、ということについても今後触れていく。

1.3 練習環境

プロコンの練習を行うに当たって、有用なサイトを紹介する。

- 会津オンラインジャッジ

  http://rose.u-aizu.ac.jp/onlinejudge/index/jsp

  プロコンで実際に出された問題を始め、オリジナルの問題など、多くの問題がある。解答の提出形式が国内予選というよりはアジア予選のものと似ており、またアジア予選と同様の制限がかけられてもいるが、肩慣らし程度の難易度の低い問題もあり、始め易いだろう。

- ACM/ICPC 国内予選突破の手引き

  http://www.deqnotes.net/acmicpc/

  国内予選突破のために解くべき問題、というのが紹介されている。ただ、解答がほとんどC++で書かれているため、参考になるのは解法の流れや考え方ぐらいである。

- ACM/ICPC OB/OG の会

  http://acm-icpc.aitea.net/

  コンテンツに模擬予選がある。

2 注意事項

プロコンの問題を解くのに必要な注意事項をここに示しておく。これを踏まえた上で傾向と対策に臨んでいただきたい。

2.1 事前準備

- 持ち込めるものは印刷物だけなので、いくつかのよく使用する関数をあらかじめ作成しておく。

- 事前に変数名のルールを作っておくなど、三人のプログラムの形式が似るような工夫が必要となる。

- 三人で一台のマシンを使う、というプログラミングへの慣れが必要で、特にプログラムの作成にどう絡むかをあらかじめ決めておくとよい。以下はその例である。
  - 問題の解析・実装で役割を分担する
  - 一人一人別の問題を解き実装する

このマニュアルは競技中に役立つように、よくある処理のテンプレート[1]やプログラムを組む際の工夫も載せてある。競技の際にはそちらを参考にしていただきたい。

2.2 コーディング前

大まかな指針を決める際に気を付けることを以下に記す。

- 最大値・最小値や、条件、使う値など、どのような制限があるか自分で書き出しておき、プログラムの大まかな仕様を定めて疑似コードを書く。変更を加えるときはなるべくそのままの方法を使う。この値や制限をおざなりにしてバグを作ってしまうこともあるので、そういった値や制限には特に気を付ける必要がある。
- **与えられる入力には不正な値はないので,不正なデータの処理は考えなくてもよい。**

  これを知っているだけでもコードを書く時間は短縮される。
- データ数の上限はたいてい示されているので、上限文だけ配列などを確保しておけばよく、動的メモリ操作はほとんど必要ない。
- 実行速度やアルゴリズムの速度はあまり気にせず、書きやすいアルゴリズムを使う。 無理に最適化しない。ただし、計算量には注意する。下手に最適化するとデバッグが難しくなり、間違いを埋め込むこともある。国内予選では実効速度の制限などはないので、あまり気にする必要はない。しかし相当時間がかかるプログラムでは必ず止まるかがあやしいので、その点は気を付けなければならない。
- **main 関数以外の関数を作るとき、グローバル変数を用いると楽になる問題もある。**
  - 引数として渡すとポインタかそうでないかで思ったような処理が実現できないことも多く、また引数も多くなってしまう。
- int 型の桁溢れを起こす用な問題もごく稀にあるので、気を付ける必要がある。

2.3 コーディング時

コーディングの際、よくあるバグを起こさないためにも必要なことを以下に記す。

- 国内予選では各自のコンピュータ上で動作させて、その出力結果をサーバに送るので、**入力は標準入力を用いる。**
- **出力は標準出力を用いる。**また出力時には改行を忘れない、無駄な空白を入れないなどに気を付ける。

[1] C++、Java の方がスタックやリストなどを簡単に書けるため、多くの対策サイト・書籍ではそれらで書かれていることが多い。しかしこのマニュアルは CS コースの学生のためのマニュアルなので、テンプレートは C 言語で書かれている。

- 一行ごと入力するときには入力データがバッファを溢れないように配慮する。
- **scanf と fgets の動作の違いに注意する必要がある。**併用するときは、改行コードを元のテキストに残さないようにするために、**scanf の% X の前後に空白を入れる。**
- 配列の番号付けと問題の番号付けが違うことがある。そのときは、0 を基準としてプログラムを作り、出力時に 1 を足すなどして変換する。※ 1 から始まるときは、最大数を増やして 0 を使わない、という方法もある。
- **グローバル変数とローカル変数をしっかりと区別する。**関数の中と外をはっきりと分ける。
- 似た名前, 入れ子、否定条件はなるべく使わない。
- 問題のサンプル入力を入れてみて、サンプル出力と合っているかを確認する。

3 バグについて

プログラムを書く際に最も煩わしいものは, デバッグであるだろう。ここではバグを未然に防ぐ方法とバグの潰し方について取り上げる。

3.1 バグを作らないために

ここでは、バグを未然に防ぐ方法について説明する。

まず始めに紹介するのは、コンパイルオプションとして-Wall を用いることである。このオプションは様々な警告を与えらてくれる。この警告を無視してもプログラムを実行できるが、大抵の場合バグが発生しセグメンテーションフォルトや答えが合わない、といったことを引き起こしてしまう。-Wall による警告が出てこなくなるまで変更を加えなければ、バグを解消することは難しい。

次に、プログラムを組む際に気を付けてほしいことを述べていく。

- ループを作る際、ループ内で用いるものを確認し、ループの外で初期化を行う。
- 変数の命名規則を決めておく。

  以下はその例である。

  - 意味の無い変数名を用いることは極力避ける。
  - int 型の汎用整数に i,j,k,l を用いる。
  - 二次元配列 A[ ][ ] にアクセスするときは A[y][x] や A[i][j] を用いる。
  - 最大数の変数は Max を末尾に付ける。
  - 回数をカウントする変数は Count を末尾に付ける。

  変数の命名規則を決めておくだけで実装中に変数名に悩むことが少なくなり、またチーム内のメンバーもデバッグに参加できる、という利点もある。実装中にその変数が何を意味しているかがわからなくなることがあるが、これも減らすことができる。

3.2　バグの潰し方

次に、バグを扱うにあたって、どのような場面でそれが出現するのかを知る必要があるが、筆者が実際にプロコンの問題を解き, 直面したバグを分類すると大きく分類すると以下のようになる。

- ループ (反復) に関するもの
- 変数名に関するもの
- アクセスに関するもの

以下ではまずこの三種類のバグに関して対処法を示す。

3.2.1　ループに関するバグ

ループに関するバグの典型的なものを挙げる。

- プログラムが実行できても止まらない、無限ループが起こる

  無限ループが発生しているような状況の対処としては、printf 文などをループの中に入れて出力してみることによって、どのような状況でループが終わらないかを判断することが出来る。ループの始まりや終わり、条件分岐の前後などに入れてみると良い。

- 多重ループの内側のループで break してループを抜けたつもりになってしまう

  多重ループから抜け出すには、通常は制御変数を導入しなくてはいけない。それによって煩雑なコードになってしまう場合などには、goto 文を用いる方法もある。ただし、コードの流れが読みにくくなってしまうことがあるので、多重ループを抜けるときのみに使うなどといった制限を各自で設ける必要がある。

  以下に goto 文の例を示す。

```
while(1){
  処理１;
  while(1){
    処理２;
    if(条件 C){
      goto L1;
    }
  }
  L1:処理３;
}
```

ループに関してのバグの多くは停止しない、というものだった。これを防ぐために、ループを出る際の条件は簡潔に書いておく必要があるだろう。そうしないとその条件がいつ満たされるかがわかりにくくなってしまうためである。特に、条件を満たすまで無限にループするような while 文は while(1) のように書いておくほうがよい。

3.2.2　変数に関するバグ

変数に関するバグは以下の通りである。

- 変数 i,j を逆に用いてアクセスした (例：A[i][j] と書くところを A[j][i] と書いた)
- 変数名を変更 (x ↑ xNum など) した際に、その変数を使用しているところを書き直し忘れる

１つ目のバグについては、テンプレートの部分である程度関数化したものがあるので、それによって大半が防がれるだろう。

また２つ目のバグに対しては、変数への命名規則を設けることでバグを未然に防ぐことができる。例については 3.1 節で触れているのでそちらを見ていただきたい。

3.2.3　アクセスに関するバグ

典型的なバグは以下の通りである。

- セグメンテーションエラー
- 二次元配列へのアクセスミス
- アクセスする範囲のミス

セグメンテーションエラーを始め、アクセスに関するミスは多く見られる。特に、スタックやリストなどの構造を用いる場合にもアクセスエラーがよく起こる。

このようなアクセスエラーは定義域をしっかり把握していない場合や、構造の中身がからであるかどうかを判別していない場合が多い。定義域を把握し、構造がからであるかどうかを判別する関数を作っておく必要があるのだが、その関数はテンプレートの章にあるのでそちらを参照していただきたい。

また、セグメンテーションエラーは起きるところが限られているので、境界条件などに気をつけることである程度は防ぐことができる。

3.2.4　その他のバグ

ここでは、先の三つに分類されなかったバグの中で、よくあるものを取り上げる。

- デバッグ用コードの消し忘れ

  デバッグ用コードの消し忘れを防止する簡単な方法としては、デバッグ用の処理コードを if 文の中に入れ、制御変数 debug を導入する、というものがある。

  その例は以下の通りである。

```
#define debug 1
...
   if(debug == 1){
      デバッグ用処理;
   }
```

このようにすることで、宣言文の定数を 1 から 0 に変えることでその処理を外すことができる。

なお、チームでデバッグをする際にもどこがデバッグ用の処理なのかを明示できる、という利点がある。

- 定数を勘違いして、修正に時間がかかった

  問題ごとに与えられる定数 (最大数など) は、コードの先頭で define 文で宣言しておくことが望ましい。これを行うと、定数を変更する際に変更箇所を少なくすることができる。

- return 文を忘れた、送る関数を間違えた

  return 文は関数を作成した際に書いておくことを習慣にすることで、書き忘れを防ぐことができる。送る関数を間違えるのを防ぐには、作った関数にプロトタイプ宣言をして入力・出力を確認し、どんな処理が行われるのかをコメントアウトしておくとよい。

バグはコンパイラが明示してくれるものとそうでないものにわかれるが、前者のものはエラーの場所を示してくれるので、間違いを見つけて直すのは難しくない。しかし、後者のものはどこが間違っているのかが判断しづらく、多くの時間を費すだろう。コンパイラオプションに-Wall を用いてある程度のバグをコンパイラに明示してもらうこともできるが、-Wall でも示せないバグも存在する。そのようなバグを、この章を参考にしてその時間を少しでも減らしてほしい。

4 傾向分析

この章では、過去の国内予選のデータを基に、国内予選突破に必要になるボーダーラインや出題傾向などの有益な情報を得ることを試みる。

4.1 突破最低ライン

突破最低ラインとは、国内予選突破のために最低限必要になるだろう正答数を意味する。過去の国内予選の結果を基に、全予選通過者のうちワイルドカード・女子枠を除いた最低正答数を調査することでこれを割り出すことを試みた。以下の図を参照していただきたい。

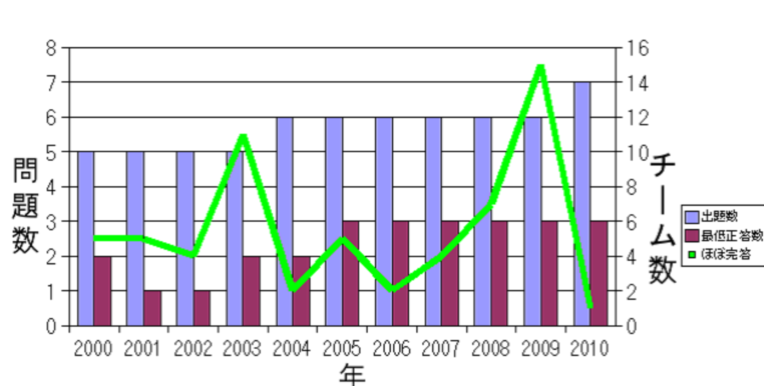

図 1: 国内予選突破最低ライン

左の棒がその年度の出題数、右の棒がその年度の国内予選突破チームの中の最低正答数であり、折れ線がほぼ完答（全問正解、あるいはあと一問で全問正解）したチームの数を表している。ほぼ完答のチーム数が劇的に多い年の翌年に、出題数が増えていることに注目していただきたい。つまり出題数を増やす意図は、出場者全体の正解数を上げることではなく、（ほぼ）全問正解者を減らし、上位陣の成績の差別化をより明確に行うことにある。たしかに最低正答数は年を経るごとに上昇する傾向が見られるが、これは上述の理由により、出題数が増加し、解けそうな問題が増えたというよりも、参加チームのレベルが向上していることが要因であると思われる。この傾向からも、最低正答数は近年の３問程度が妥当だろう。３問ではタイムペナルティによっては突破することができていない年もあるが、ここではまず３問を確実に解くことを目標とし、４問目以上の対策については後の課題とする。

4.2 アルゴリズムの出題傾向

出題傾向を把握することによって、特に対策を打つべきアルゴリズムを見つけ出し、それらのアルゴリズムを優先的に練習を行うことで、より効率良く、より確実な突破を目指すことができると考えた。以下がアルゴリズムの出題傾向を示すグラフである。

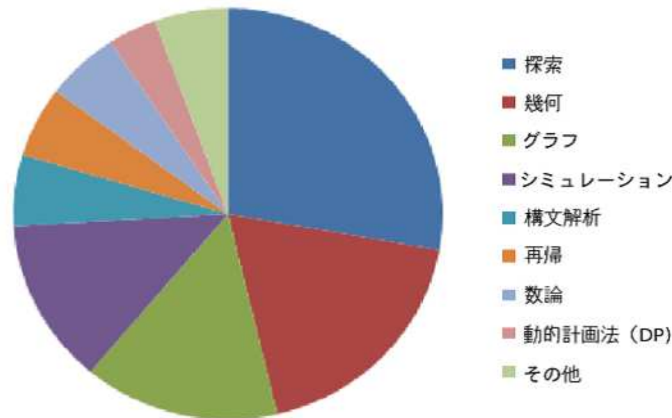

図 2: 2000〜2010 年のアルゴリズムの出題傾向

このように、国内予選では特に探索、幾何、グラフ、シミュレーションの問題が多く出題されていることがわかる。しかし、以下のグラフにも目を通していただきたい。

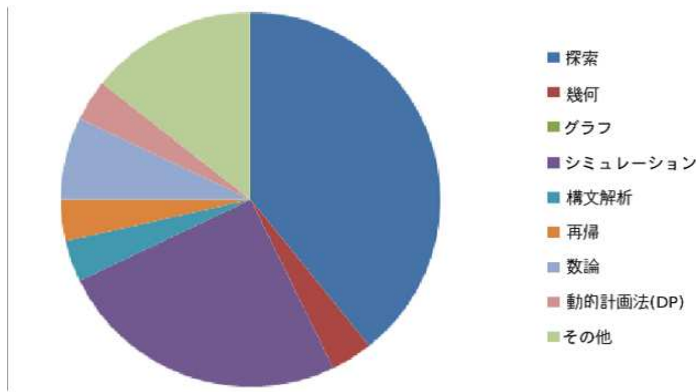

図 3: 2000〜2010 年の予選突破最低ラインの出題傾向

これは、それぞれの年の問題を、参考サイト等の情報から平易と思われる順に問題を並べ、突破最低ラインである３題を取り出して見たときのデータである。突破最低ラインにおいては、全問題での傾向に比べ、幾何・グラフの問題が占める割合が圧倒的に減少していることが見てとれる。予選突破を第一と考えるのであれば、この図で頻出とされている探索・シミュレーションの問題に対する対策が最優先といえるだろう。

しかし、これだけでは突破最低ラインの３題分には届かないため、あとふたつ程度のアルゴリズムを軽く取り上げたいと思う。

ひとつめに、数論を取り上げたい。理由としては、図３のなかで３番目に頻出であること、また、数論の問題は多くが素数を扱う問題であり、素数表のテンプレートを用いると簡単に解ける可能性が高いことなどが挙げられる。ふたつめは、動的計画法（DP) である。出題頻度はあまり高くないが、国内予選突破の手引きで特集が組まれる、国内模擬予選で多く出題されるなど、頻出傾向は高まると考えれる。

4.3　データ構造の頻出傾向

データ構造に関しても同様に、国内予選突破最低ラインの問題に対して、頻出傾向を分析した。それが以下の図である。

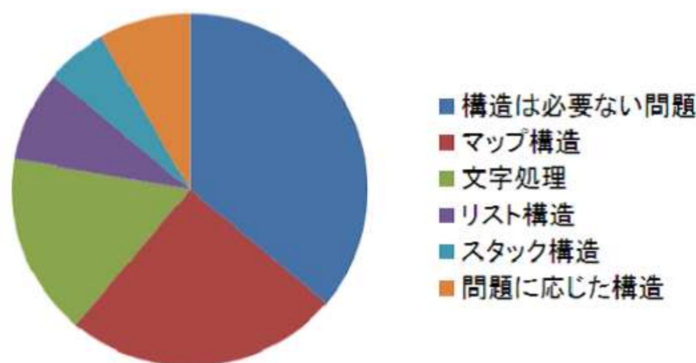

図 4: 2000〜2010 年の予選突破最低ラインのデータ構造

構造が必要ないとは、入力を逐次処理する、あるいは、配列などに番号付けして格納しておくだけで充分であるという意味で、データ構造に工夫を要さない、ということである。よって、対策すべきはマップ構造や文字列処理であるとわかる。マップ構造とは、主に２次元平面を想定する問題において、座標が整数で表せるときに、平面の様子を再現するのに用いる構造である。文字列処理では、配列の配列を扱うケースがある、NULL 文字・改行文字などに例外処理を施す必要がある、string.h の関数を利用する必要があるなど、問題によっては複雑な操作になるため難しくなることも多い。他にも、リスト、スタックなどが上がっているが、これらに関してはテンプレートを事前に用意し、使いかたを把握しておけば、楽に問題を解くことができる。

4.4 毎年の出題傾向

表 1: 毎年の突破最低ラインの出題傾向

	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	計
探索	2	2	1	1	1	1	2		1	2	1	14
シミュレーション	1			1	1	1		2		1	1	8
構文解析			1						1			2
数論							1		1			2
再帰					1							1
DP				1							1	2
その他		1	1			1		1				4

上の表は、突破最低ラインの問題構成をまとめたものである。やはり探索とシミュレーションが多いことがわかるが、それだけでなく、シミュレーションと探索の両方が出題されない年がなかったこともみてとれる。また、2006 年以降のデータに注目すると、シミュレーションがない年には代わりに数論が出題されているとわかる。すなわち、探索・シミュレーション・数論の３種の問題をこなせていれば、それだけで２〜３問の正答を稼げていたことになる。

5　対策と例題

この章では、前章の分析の結果を受け、探索・シミュレーション・数論・動的計画法の問題について解説する。

ただし、問題のサンプル入力とそれに対する出力は、hanafuda shaffle 以外省略する。サンプル入力とそれに対する出力は、会津のオンラインジャッジ [2]や ACM-Japan [3]で手に入るので、そちらを見ていただきたい。

5.1　シミュレーション

- 概要

  シミュレーションには明確にアルゴリズムは存在せず、問題の指示に従った操作をソースコードで忠実に再現することにより解ける問題である。

- 見分けかた

  シミュレーションの問題では、問題文には入力と出力、そしてその変換に必要な操作を明記してある。このような問題の中には数学やアルゴリズムの知識を用いることで計算量を抑えることが可能なものも多いが、その計算量が現実的でなくなる場合を除いて、通常は愚直にシミュレーションを行う方がコーディングが容易かつわかりやすくなる。つまり、行うべき操作方法が示されている問題であれば、データのサイズが膨大でない限り、まずはシミュレーション問題だと考えて取り組むのが良い。

以下にシミュレーション問題の例題と、解答ソースコードを示す。ソースコードは一例にすぎないので、よりわかりやすい、あるいは省計算量な解答を自分でも考えてみてほしい。

5.1.1　Hanafuda Shuffle (ACM-ICPC Japan Domestic, 2004-07-02)

問題

カードの山を混ぜて切る方法はいろいろとある。日本のカード遊びである「花札」における花札混ぜ切りは，札を切る方法の一つ である以下に，花札混ぜ切りの方法を示す。

n 枚のカードの山がある。図 1 に示すように，山の一番上から p 枚 目の札から連続した c 枚の札を抜き取り，それをそのまま山の上に置く． この操作 (カット操作という) を繰り返し行う。花札混ぜ切りをシミュレートし，最終的に山の一番上にくる札を答えるプログラムを書きなさい。

[2]http://rose.u-aizu.ac.jp/onlinejudge/index/jsp
[3]http://www.acm-japan.org/

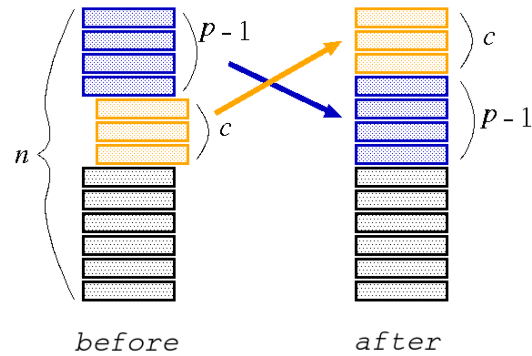

図 5: cutting opperation

- Input

  入力は複数のデータセットから構成される。各データセットは n と r という二つの正の整数を含む行から始まる。ただし，n は $1 \leq n \leq 50$ であり，r は $1 \leq r \leq 50$ である． n と r はそれぞれ山にある札の枚数とカット操作の回数を表す。データセットはさらに r 行続く．その各行は 1 回のカット操作を表して おり，これらの操作は順に実行される。各行は p と c の二つの正の整数を含む。ただし，p と c は $p + c \leq n + 1$ を 満足している。札の山の一番上から p 枚目の札から，c 枚の札を山から抜 き取り，その山の一番上に置く。入力の終わりは 0 を二つ含む行によって表される。各入力行は一つの空白で区切られた二つの整数を含む。行内にその他の文字はない。

- Output

  入力の各データセットに対して，最初に一番下を 1 番として順に n 番まで の札が積み上げられた山を仮定して，山を混ぜて切り終わったとき，山の一番 上にある札の番号を出力しなさい。ただし，番号は前後に空白のような余分な文字を含まないこと。

- Sample Input

```
5 2
3 1
3 1
10 3
1 10
10 1
8 3
0 0
```

- Output for the Sample Input

```
4
4
```

解説

配列を用いることで、カードの山を再現することができる。操作の再現は、

1. 指定された部分のカードを抜き取る　↑　保存用の配列にいったん保存
2. 山の上の部分を下に詰める　↑　配列の要素をずらして格納しなおす
3. 山に抜き取ったカードを乗せる　↑　保存用の配列から元の配列の上に格納しなおす

という 3 ステップに分け、実現すればよい。

解答例

```
#include<stdio.h>

int main(void){
  int i,j;
  int n,r;
  int p,c;
  int h[50],tmp[50];

  while(1){
      scanf("%d %d",&n,&r);
      if(!n && !r)break;

      for(i=0;i<n;i++)h[i] = n-i;      //一番上の札を配列の 0 番目とする

      for(i=0;i<r;i++){
         scanf("%d %d",&p,&c);
         p--;   //一番上の札を 1 でなく 0 にしているため, ズレを修正

        //p-1 番目の札から c 枚を tmp に保存 (抜き取り)
         for(j=0;j<c;j++)tmp[j] = h[j+p];

        //上から p-1 枚の札を、下の札と合わせる
         for(j=p-1;j>=0;j--)h[j+c] = h[j];

        //tmp に保存していた札をさらに上に重ねる
         for(j=0;j<c;j++)h[j] = tmp[j];
      }
      printf("%d\n",h[0]);
    }
  return 0;
}
```

5.1.2 Next Mayor (ACM-ICPC Japan Domestic,2009-)

問題

芸無町の奇妙な風習のひとつは，次期町長の選出までもがゲームの結果によることだろう。町長の任期満了が近づいてくると，現職の町長を含む少なくとも 3 人の候補者が小石のゲームを競い，勝者が次期町長となる。

小石のゲームのルールは次の通りである．以下で，n は候補者の人数である。

- 使うもの

  円卓と碗と十分な個数の小石.

- ゲームの開始

  最初に碗に入れるのは管理委員会が秘密の確率的手段で決めた数の小石である。0 から n-1 と番号を振った全候補者は，反時計回りの番号順に円卓を囲んで座る。碗はまず現職の町長 (候補者 0) に渡す。

- ゲームのステップ

  碗を渡された候補者は，碗に小石が入っている場合は，そのうち 1 個を取り（すでに持っている小石があればそれらと共に）手許に置く。碗が空の場合は，手許に小石があればその全部を碗に入れる。どちらの場合も，その後で碗を右隣の候補者に渡す。勝者が決まるまでこのステップを繰り返す。

- ゲームの終了ある候補者が碗に残った最後の小石を取り出したとき，他のどの候補者の手元にも小石がなければ，ゲームは終わりで，全部の小石を持っているその候補者が勝者となる。芸無高校の数学教員の解析によると，このゲームは必ず有限のステップで終わるが，必要なステップ数は非常に多くなることもある。

- Input

  入力ははデータセットの並びである。各データセットは一文字の空白で分けられたふたつの整数 n と p からなる一行である．整数 n は現町長を含む候補者の人数であり，整数 p は最初に碗に入れる小石の総数であり、$3 \leq n \leq 50$, $2 \leq p \leq 50$ である。入力データセットに含まれる設定では，ゲームは 1000000(百万) ステップ以内に終了する。入力の終わりは，ふたつの 0 が一文字の空白で区切られる一行で示される。

- Output

  出力は，入力の各データセットの表すゲームの勝者の候補者番号を示す整数ひとつからなる行を，入力データセットの順序どおりにならべたものである．それ以外の文字が出力にあってはならない.

解説

椀の中の小石の数を表す変数を作り、まず、用意された小石の数を記憶させる。
最初の候補者０から、

椀に小石があれば　↑　椀の小石をひとつ減らし、候補者の小石をひとつ増やす
椀に小石がなければ　↑　椀の小石の数に、候補者の小石の数をコピーする
（候補者が小石を持っていなければ、ゼロがコピーされる）

として、終了条件を満たすまで候補者をずらしていく。椀が空で、他の候補者がひとつも小石を持っていないが一人だけ持っているとは、（ある候補者が持っている小石の数＝用意された小石の数）になっていることに等しい。よってこれを終了条件に使うことができる。

解答

```
#include<stdio.h>

int main(void){
  int i;
  int a[50];
  int n,p,c;

  while(1){
      scanf("%d %d",&n,&p);
      if(!n && !p) break;
      c = p;
      for(i=0;i<n;i++) a[i] = 0; //候補がもつ小石の数を初期化

      i = 0;
      do{
        if(i==n)i=0;  //最後の人を越えたら最初の人に戻る
        if(c>0){     //椀に小石があるとき
          a[i]++;    //候補は小石をひとつ取り
          c--;       //椀の小石が一つ減る
        }else{       //椀に小石がないとき
          c = a[i]; //候補の小石を椀に移し
          a[i] = 0; //候補の小石はなくなる
        }
        i++;
      }while(a[i-1]<p); //候補がすべての小石を持っていたら break
      printf("%d\n",i-1);
    }
  return 0;
}
```

5.2 探索

- 概要

  探索には、最大値・最小値を求める問題や、条件に一致するものがないかを調べる、あるいはその個数を数え上げる問題などがある。アルゴリズムとしては、深さ優先探索と幅優先探索があるが、応用としてバックトラック法を用いることも多い。突破最低ライン問題の探索では、想定されうる全ての場合を調べること（全探索）が基本となる。

- 見分け方

  すべてのパターンを調べることで答えが得られるような問題は基本的に探索である。また、パズルなど盤面の状態変化を扱う問題で、すべてのパターンを調べるために前の状態を復元する必要がある場合は、バックトラック法を用いて解く。

以下に探索問題の例題と、解答ソースコードを示す。ソースコードは一例にすぎないので、よりわかりやすい、あるいは省計算量な解答を自分でも考えてみてほしい。

5.2.1 Get Many Persimmon Trees

問題 (原文は英語なので、以下は筆者による日本語訳である)

林誠司 (ハヤシセイジ) は、１８世紀の間、永きに渡って会津領の日新館 (侍学校) で教授を勤めました。教育における彼の立派な功績に報いるため、会津藩の藩主である松平容頌 (マツダイラカタノブ) は、会津領内の広大な土地の中から、長方形の土地を彼に譲渡することを決めました。土地の大きさ (幅、および高さ) は藩主により厳密に指定されましたが、指定された土地のなかであれば、自由に土地を選ぶことができました。また長方形の土地には、会津の名産のひとつである "見知らず柿" という柿の木が植えてありました。柿は林の大好きな果物だったので、彼は与えられる土地の中にできる限り多くの柿の木が欲しいと考えました。

例えば、図 6 では、全体の土地は幅と高さがそれぞれ 10 と 8 である格子状の長方形です。'*' は柿の木の場所を示しています。指定された幅と高さがそれぞれ 4 と 3 であれば、実線によって囲まれた領域が最も多くの柿の木を含んでいます。同様に、破線によって囲まれた領域は、幅が 6 であり高さが 4 である長方形のうち一番柿の木が多く、点線によって囲まれた領域は、幅と高さがそれぞれ 3 と 4 であるときには、最も多くの柿の木を含む領域です。幅と高さは区別されることに注意してください。サイズ 4×3 と 3×4 は、図 1 に示されているように、別の扱いです。

図 6: Expamles of Rectangular Estates

あなたの仕事は与えられた大きさ (幅と高さ) の土地のうち、もっとも多くの柿の木を含むような土地を探すことです。

- Input

  入力は複数のデータセットから成ります。各データセットは以下の様式で与えられます。

  N

  W H

  $x_1 y_1$ $x_2 y_2$ ... $x_N y_N$

  S T

  N は柿の木の数であり、５００以下の正の整数です。W および H はそれぞれ、全体の土地の幅と高さを表しています。W と H はともに正の整数であり、１００以下の値をとると考えてよいです。各々の i $(1 \leq i \leq N)$ について、x_i と y_i は i 番目の柿の木の座標を表しています。座標の開始が 1 からであることには注意してください。$1 \leq x_i \leq W$ かつ $1 \leq y_i \leq H$ であり、ふたつの木が同じ座標にあることはないものとします。しかし、座標に応じて柿の木の順番が決まっているとは限りません。最後に、寄贈される土地の幅、および高さとして、S と T が与えられます。$1 \leq S \leq W$ 、$1 \leq T \leq H$ です。入力の終わりはひとつのゼロによって表されます。

- Output

  各データセットに対して、与えられたサイズの土地のうち、含んでいる柿の木の数がもっとも多くなるように土地を求め、その柿の木の最大数を出力しなさい。

解説

全体の土地を左上から右下へ、とりうるすべての長方形について、囲う柿の木の数を確かめていけばよい。このとき、長方形の左上の座標をずらしていくことですべての場合を調べられるが、右や下が全体の土地の範囲を越えると、セグメンテーション違反を起こす可能性があるので、注意しなければならない。

解答例

```
#include<stdio.h>

int main(void){
  int i,j;
  int n;
  int w,h;
  int map[100][100];
  int x,y;
  int s,t;
  int sum,max;

  while(1){
    scanf("%d",&n);
    if(!n)break;

    scanf("%d %d",&w,&h);

    for(i=0;i<h;i++){
      for(j=0;j<w;j++){
      map[i][j] = 0;        //mapの初期化
      }
    }

    for(i=0;i<n;i++){
      scanf("%d %d",&x,&y);
      map[y-1][x-1] = 1;     //木のあるところをcheck
    }

    scanf("%d %d",&s,&t);    //長方形の幅がs、高さがt

    max = 0;
    /*長方形の左上の座標を動かして、全探索*/
    for(y=0;y<=h-t;y++){     //下端が最下端(h-1)のとき、上端はh-t
      for(x=0;x<=w-s;x++){  //右端が最右端(w-1)のとき、左端はw-s
        sum = 0;
        for(i=y;i<y+t;i++){
          for(j=x;j<x+s;j++){  //長方形中の各座標について、
            sum+=map[i][j];     //木があれば総数を+1
          }
        }
        if(max<sum)max = sum;
```

```
      }
    }

    printf("%d\n",max);
  }
  return 0;
}
```

5.2.2 Curling 2.0

MM-21 星でもオリンピック以来カーリングが流行している。しかし, そのルールは地球のものとはすこし異なっており, マス目状の氷の盤上で石を一つだけ使って行われる。スタート位置からゴール位置まで石を到達させる移動回数の少なさを競うのである。

図 D-1 に盤面の例を示す。盤上のマス目には障害物が配置されていることがある。盤面には, スタートとゴールという特別なマスが一つずつあり, そこには障害物はない。(スタートとゴールが一致することはない。) 滑りはじめた石は障害物がないかぎりどこまでも進んでいくので, ゴールに到達させるには, 障害物を利用していったん石を止め, あらためて滑らせてやる必要もあろう。

図 7: D-1:盤面の例 (S:スタート、G:ゴール)

石の動きは以下の規則に従う：

- ゲーム開始時に、石はスタートで止まっている
- 石の動きは x,y 方向に限る。ななめには動けない。
- 止まっている石は、滑らせることによって動き出す。その時の方向は, すぐ次のマスに障害物が無い方向ならどちらでもよい (図 D-2(a))。
- 動き出した石は, 次のいずれかが起こるまで, 同じ方向に動き続ける。
  - 障害物にぶつかった場合 (図 D-2(b), (c))。
    - 石は障害物の一つ手前のマスで止まる。
    - 障害物は消滅する。
  - 盤外に出た場合,

        * 失敗でゲームは終わる。
    - ゴールの上に来た場合,
        * そこで石は止まり, 成功でゲームは終わる。

- 1 回のゲーム中の滑らせる回数の最大は 10 である。この回数でゴールに石を到達させることができない場合, 失敗でゲームは終わる。

図 8: D-2:石の動きの例

以上の条件のもとで, スタート位置にある石をゴール位置に到達させることができるか, できるなら最小何回滑らせればよいかを知りたい。

図 D-1 に示す初期配置では 4 回で石をスタート位置からゴール位置に動かすことができる。そのときの経路を図 D-3(a) に示す。 なお, 石がゴールに到達した時点での盤面は図 D-3(b) のように変化している。

図 9: D-3:図 D-1 の解と終了時の盤面

- Input

  入力はデータセットの並びである. 入力の終わりは, 一つの空白で区切られた二つのゼロからなる行で示される。データセット数が 100 を超えることはない。

  各データセットは次のような形式をしている:

0	何もないマス
1	障害物
2	スタート位置
3	ゴール位置

```
盤の幅（=w）盤の高さ（=h）
盤面の 1 行目
...
盤面の h 行目
```

盤の幅と高さは次の条件を満たす: $2 \leq w \leq 20, 1 \leq h \leq 20$

盤面の各行には, w 個の数字が空白一つをはさんで並んでいる。その数字は対応するマス目の状態を示している。

図 D-1 に対応するデータセットの記述は以下のようになる:

```
6 6
1 0 0 2 1 0
1 1 0 0 0 0
0 0 0 0 0 3
0 0 0 0 0 0
1 0 0 0 0 1
0 1 1 1 1 1
```

- Output

  各データセットが指定する盤面について, スタート位置にある石をゴール位置に到達させるまでに滑らせる回数の最小値を, 十進の整数値でそれぞれ 1 行に出力せよ。そのような移動ができない場合には, -1 を出力せよ。各出力行はこの数値以外の文字を含んではならない。

解説

実際に起こり得る石の動きをすべて再現すれば、10 回以内にゴールにたどり着けるのか、また、たどり着く最小の試行回数はいくらなのかを求められる。これは、上下左右それぞれについて、

1. 移動できるかを判定。(隣が障害物でないか？)
2. ゴールがあれば終了。
3. 障害物があれば、その手前に石を動かし、障害物を壊す。
4. 場外まで出てしまう場合、なにもしない

ということを繰り返していけばよい。そのためには、これらの試行を行う関数を作って、それを試行が終わる度に再帰的に呼び出せばよい。

しかし、単純に再起で再現しようとすると、一度壊した障害物が壊れたままになり、答えが変わってしまう。ここで、バックトラック法を用いる。すなわちこの問題の場合、

```
  block = 0;  pos = new;   //障害物の破壊、石の移動
  recursive();             //関数の再帰呼び出し
  block = 1;   pos = pre;  //障害物の復元、石の位置の復元
```

というように、一度新しいデータに更新してから関数を再帰呼び出しして、その後また前の状況を復元してやる。これにより、再帰呼び出しを行う前と後で、同じ盤面を保つことができる。このバックトラック法は他にも、パズルを解く問題などの探索問題で力を発揮する手法である。

解答

```
#include<stdio.h>

int w,h;
int a[20][20];

int curling(int sh,int sw,int n){
  int i,min,tmp;

  min = 11;           //最小値を上限越えの 11 に設定
  if(n>10)return n;  //10 回以内にゴールにたどり着けなかった場合、11 を返す
  else{
    if(sw-1 >= 0 &&a[sh][sw-1]!=1){  //左に石を滑らせることが可能な場合
      for(i=sw-1;i>=0;i--){
        if(a[sh][i]==3)return n;     //ゴールにたどり着けた場合、その試行回数を返す
        if(a[sh][i]==1){             //障害物に当たったとき
          a[sh][i]=0;                //障害物の破壊
          tmp = curling(sh,i+1,n+1); //石の位置と試行回数を更新し、再帰呼び出し
          if(min > tmp)min = tmp;    //最小値の更新
          a[sh][i]=1;                //障害物の復元
          break;                     //試行の終了
        }
      }
    }

    if(sw+1<w &&a[sh][sw+1]!=1){     //右に石を滑らせることが可能な場合
      for(i=sw+1;i<w;i++){
        if(a[sh][i]==3)return n;
        if(a[sh][i]==1){
          a[sh][i]=0;
          tmp = curling(sh,i-1,n+1);
          if(min > tmp)min = tmp;
          a[sh][i]=1;
          break;
        }
```

```
      }
    }

    if(sh-1>=0 && a[sh-1][sw]!=1){    //下に石を滑らせることが可能な場合
      for(i=sh-1;i>=0;i--){
        if(a[i][sw]==3)return n;
        if(a[i][sw]==1){
          a[i][sw]=0;
          tmp = curling(i+1,sw,n+1);
          if(min > tmp)min = tmp;
          a[i][sw]=1;
          break;
        }
      }
    }

    if(sh+1<h && a[sh+1][sw]!=1){      //上に石を滑らせることが可能な場合
      for(i=sh+1;i<h;i++){
        if(a[i][sw]==3)return n;
        if(a[i][sw]==1){
          a[i][sw]=0;
          tmp =  curling(i-1,sw,n+1);
          if(min > tmp)min = tmp;
          a[i][sw]=1;
          break;
        }
      }
    }
  }

  return min;
}


int main(void){
  int i,j;
  int sw,sh;
  int ans;

  while(1){
    scanf("%d %d ",&w,&h);
    if(!w && !h)break;
```

```
    for(i=0;i<h;i++){
      for(j=0;j<w;j++){
        scanf("%d ",&a[i][j]);
        if(a[i][j]==2){
          sh=i;        //石の初期位置をメモ
          sw=j;
        }
      }
    }

    ans = curling(sh,sw,1); //最小の回数を求める関数により、答えを求める

    if(ans>10)printf("-1\n");  //関数は 10 回以内に終わらない場合、11 を返す
    else printf("%d\n",ans);
  }
  return 0;
}
```

5.3 数論

- 概要

  数論とは、数学的な知識を利用することによって解答することができる問題を指す。予選突破ラインの問題においては、そのほとんどが素数を扱う問題である。

- 見分け方

  素数や最小公約数など、数学的な知識を要求する問題は数論に属する。

以下に数論の問題の例題と、解答ソースコードを示す。ソースコードは一例にすぎないので、よりわかりやすい、あるいは省計算量な解答を自分でも考えてみてほしい。

5.3.1 Dirichlet's Theorem on Arithmetic Progressions

こんばんは，選手諸君。

a と d が互いに素な正の整数ならば， a から始まり d ずつ増える等差数列 $a, a+d, a+2d, a+3d, a+4d, ...$ には無限個の素数が含まれる。このことはディリクレの算術級数定理として知られている。ガウス (Johann Carl Friedrich Gauss, 1777 - 1855) が予想していたことを， 1837 年にディリクレ (Johann Peter Gustav Lejeune Dirichlet, 1805 - 1859) が証明した。

たとえば， 2 から始まり 3 ずつ増える等差数列

```
2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50,
53, 56, 59, 62, 65, 68, 71, 74, 77, 80, 83, 86, 89, 92, 95, 98, ...
```

は，無限個の素数

```
2, 5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89, ...
```

を含む。
そこで君の使命だが，与えられた正整数 a と d と n に対して，この等差数列に含まれる n 番目の素数を求めるプログラムを書くことにある。
例によって，君もしくは君の仲間が疲れはて，あるいは混乱しても，当局はいっさい関知しないからそのつもりで。なお，この審判システムは 3 時間で自動的に停止する。成功を祈る。

- Input

  入力はデータセットの並びである。データセットは， 1 文字の空白で区切られた三つの正整数 a と d と n とからなる行である。a と d は互いに素である。$a \leq 9307$ かつ $d \leq 346$ かつ $n \leq 210$ と仮定してよい。入力の終わりは，1 文字の空白で区切られた三つのゼロからなる行で示される。これはデータセットではない。

- Output

  出力は入力データセットと同数の行で構成されなければならない。各行は一つの整数を含まなければならない．余計な文字を含めてはならない。

  データセット a, d, n に対応する出力の整数は， a から始まり d ずつ増える等差数列に含まれる素数のうちで n 番目のものでなくてはならない。

  なお，この入力条件の下で，結果は必ず 10^6 (百万) より小さいことがわかっている。

解説

素数を扱う問題では、主にふたつの素数データを用いる。ひとつが素数表であり、もうひとつが素数配列である。
素数表とは、素数であるかどうかを True or False で判定できるようにしたものである。すなわち、

```
Prime_table[2] = true ;
Prime_table[3] = true ;
Prime_table[4] = false;
…
```

といったようになる。これを用いれば、`Prime_table[i]` とするだけで、i が素数かどうかを判別できるため、ある数が素数かどうかを知りたいときに役立つ。
これに対し、素数配列とは、素数を小さい順に格納した配列である。すなわち、

```
Prime[0] = 2;
Prime[1] = 3;
Prime[2] = 5;
Prime[3] = 7;
…
```

といったようになる。これは、素数そのものが必要になるときに利用すると便利である。
素数表を実現するアルゴリズムがエラトステネスの篩である。アルゴリズムの正当性については説明を割愛するが、これは、

1.Prime_table[0] と [1] を除き、すべて Prime_table[i] = true とする。
2.i = 0 から始めて、Prime_table[i] が true になるまで、i を増やしていく。
3.Prime_table[i] = true になったら、その i の倍数をすべて false にしていく。
すなわち、
Prime_table[2*i] = false;
Prime_tabel[3*i] = false;
…
4. 求めたい上限の値の平方根まで、2.〜3. を繰り返し続ける。

というものである。
素数配列については、この素数表を用いることで作成できる。すなわち、

```
count = 0;
for(i=0;i<MAX;i++){
  if(Prime_tabel[i] == true){
    Prime[count] = i;
    count++;
  }
}
```

とすればよい。
今回の問題では a+i*d が素数かを調べればよいので、素数表を作るだけで十分である。

解答

```
#include<stdio.h>
#define MAX 1000000

int main(void){
  int i,c;
  int a,d,n;
  char p[MAX];

  p[0]=0;
  p[1]=0;                         //1 は素数でない
  p[2]=1;                         //2 は素数である
  for(i=3;i<MAX;i+=2)p[i]=1;    //奇数は素数の可能性がある
  for(i=4;i<MAX;i+=2)p[i]=0;    //偶数は素数の可能性がない

  c = 3;
  while(1){
    for(i=2*c;i<MAX;i+=c)p[i] = 0;    //自身の倍数の check をはずす
    while(!p[++c]);                   //次の素数まで進める
    if(c>1000)break;                  //MAX の２乗根まで調べれば充分
  }
```

```
  while(1){
    scanf("%d %d %d ",&a,&d,&n);
    if(!a && !d && !n)break;
    c = 0;
    i = 0;
    while(1){
      if(p[a+i*d])c++;  //a+i*dが素数であれば、カウントを増やして個数を数え上げ
      if(n==c){              //個数がnと一致すれば
        printf("%d\n",a+i*d);  //その値を出力
        break;
      }
      i++;
    }
  }
  return 0;
}
```

5.4 動的計画法

- 概要

  動的計画法 (以下 DP) とは、最大値などの最適解を求める際に、よりサイズの小さい問題の答えを用いて最適解を導き出す手法である。DP は慣れるまでは理解しがたいアルゴリズムであるが、ある程度の数をこなすと、どういったときに使うのか、どのように使うのか、などのコツが掴めてくる。

- 見分け方

  DP の問題は、一見すると探索の問題に見えがちである。しかし、愚直に探索として解こうとすると、計算時間が間に合わなくなってしまうため、DP を用いる。探索と DP の違いを問題から読み取る際のポイントは、

  – 探索に関わる要素が複数存在する (実装すると多重ループになる)
  – 探索に関わる要素のデータサイズが膨大 (100000 以上)
  – 最大値や最短路など、最適解を求める問題である

  などがある。

以下に DP の問題の例題と、解答ソースコードを示す。ソースコードは一例にすぎないので、よりわかりやすい、あるいは省計算量な解答を自分でも考えてみてほしい。

5.4.1 Problem B Balloon Collecting(ACM-ICPC Asia Regional Contest in Tokyo, 2010-12-12)

問題

風船は効率よく取らなくてはならない、とゲームデザイナーは言う。彼は2次元グラフィックの昔ながらのゲームを設計している。そのゲームでは、風船は地面に一つずつ落ちてきて、プレイヤーは地上で車型の機械を操り風船を捕まえる。プレイヤーはその車を右か左、あるいはその場所に留まるように操作することができる。風船が地面につくとき、車と風船は同じ場所にいなければならないが、そうでない場合は風船が割れてゲームが終了する。

ゲームクリアの条件はすべての風船をゲームフィールドの左端にある家にしまうことである。車は一度に三つの風船を運ぶことができるが、そのスピードは運んでいる風船の数によって変化する。車がk個の風船を運んでいるとき(k=0,1,2,3)、距離1を動くのにk+1の時間がかかる。プレイヤーは、車の総移動距離が小さいほうが高得点を得られるだろう。

あなたの仕事はゲームデザイナーが風船のセットのゲームデータをチェックするのを助けることである。現在位置(家からの距離)と風船が地面に付くまでの時間が与えられ、プレイヤーが全ての風船を取れるかを判定し、全ての風船を取ってしまうまでのに必要な最小の移動距離を答えなければならない。もしプレイヤーが全ての風船を取れないなら、あなたはプレイヤーが取れない最初の風船を知らせなければならない。

- 入力

  入力は複数のデータセットで与えられる。各データセットの形式は次の通りである。

  n

  $p_1 t_1$

  ...

  $p_n t_n$

  最初の行には風船の数nが与えられる。nは$0 < n < 40$の整数である。続くn行には二つの整数p_iとt_i($1 \leq i \leq n$)が空白で区切られて与えられる。p_iとt_iはi番目の風船が地面に付く場所とその時間で、p_iは$0 < p_i \leq 100$、t_iは$0 < t_i \leq 50000$である。$i < j$に対し$ti < tj$と想定してよい。家の位置は0であり、ゲーム開始時の時間は0である。

  車や家、風船の大きさは十分に小さいものとし、無視して構わない。車が風船を取る、あるいはそれらを家にしまうのにかかる時間は0である。車はそれらの操作を行った後、ただちに動き始めることができる。

  入力の終わりはひとつの0からなる行で表される。

- Ounput

  各データセットに対し、出力は一語と一つの整数を空白で区切って一行で出力せよ。他の余計な文字は出力内に現れてはいけない。

  – もしプレイヤーが全ての風船を取れるなら、“OK”という文字列と車が全ての風船を取るのに必要な最小の移動距離を示す一つの整数を出力せよ。

  – もしプレイヤーが全ての風船を取るのが不可能なら、“NG”という文字列とデータセット内のプレイヤーが取れない最初の風船であるk番目の風船の番号を示す一つの整数を出力せよ。

解説

この問題はアジア予選の問題であるが、その中でも簡単で解けるものなので、ぜひとも挑戦していただきたい。

この問題で厄介なのは、自分の手元にあるバルーンの数がいくつであっても、家に置きにいくことができるという点である。置きに行ってから取りにいくか、置きにいかずに直接取りにいくかを毎回分岐させると、調べるパターンは最悪、(２の balloon 乗) になる可能性がある。これを避けるために、DP を用いる。

DP の要素にするものはふたつ、"何番目のバルーンか"と"バルーンを取り終わった後に持っているバルーンの個数"である。つまり、m 番目のバルーンを取ったとき、n 個のバルーンを持っている状態のうちの最短移動距離を DP[m][n] に格納しておけばよい。

こうすることで、DP[m][n] の値は、DP[m-1][1 or 2 or 3] を参照することにより求めることができる。すなわち、バルーンを置きに戻らないとき、$DP[m][n] = DP[m-1][n-1] + dis$ であり、置きに戻るとき、$DP[m-1][1] = min(DP[m-1][1], DP[m-1][2], DP[m-1][3]) + homedis$ である。ここで、dis は現在地から次のバルーンの落下位置までの距離であり、homedis は（現在地から家までの距離）+（家から次のバルーンの落下位置までの距離）である。この DP の計算量は $O(m^n)$ であり、全探索するより遥かに高速に終了できる。

DP を扱う際に一番難しいのは、何を DP の要素に選択するかである。これは違う要素を選択しても正答できたり、選択した要素によって計算時間が変わってしまう場合もある。どのように要素を選択していくかは重要な課題であり、ある程度の数をこなして感覚を養う努力も必要になる。

解答

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define INF 10000 //移動距離の最大値以上の数
#define BalMax 3  //持てるバルーンの最大数
#define FallMax 40//落ちてくるバルーンの最大数

//グローバル変数
int DP[BalMax][FallMax];    //DP 行列
int n,p[FallMax],t[FallMax];
char JDG[2];      //判定文（NG or OK）

//DP 初期化関数
void initDP(void){
  int i,j;
  for(i=0;i<BalMax;i++){
    for(j=0;j<FallMax;j++){
      DP[i][j]= INF;
    }
  }
}

//二項の最小値出力関数
```

```
int minimum(int a, int b){
  if(a<b)return a;
  else   return b;
}

//バルーンを家に持ち帰ってから取りにいく際の値の計算
int calc1(int num){
  int dt,dx;
  int A,B,C;
  dt = t[num] - t[num-1];//時間差
  dx = p[num] + p[num-1];//移動距離

  if(p[num]+p[num-1]*4 > dt)A = INF;  //3 個持ってる状態からの移動の判定
  else A = DP[2][num-1] + dx;

  if(p[num]+p[num-1]*3 > dt)B = INF;  //2 個持ってる状態からの移動の判定
  else B = DP[1][num-1] + dx;

  if(p[num]+p[num-1]*2 > dt)C = INF;  //1 個持ってる状態からの移動の判定
  else C = DP[0][num-1] + dx;

  return minimum(A ,minimum(B ,C) );//3 パターンの最小値の出力
}

int calc2(int bal, int num){//そのまま取りにいく際の値の計算
  int dt,dx;

  dt = t[num] - t[num-1];       //時間差
  dx = abs(p[num] - p[num-1]);  //移動距離距離

  if(dx*(bal+1) > dt)return INF;    //取れない
  else return DP[bal-1][num-1]+dx;  //左上の値 + 移動距離
}

int val(int bal, int num){
  int k;
  switch(bal){
  case 0://置いてから取りにいく場合
    k = calc1(num);
    break;
  case 1://1 個持ってる状態からそのまま取りにいく
    k = calc2(bal,num);
    break;
```

```
  case 2://2こ持ってる状態からそのまま取りにいく
    k = calc2(bal,num);
    break;
  }
  return k;
}

int solve(void){
  int i,j,min;
  if(p[0]>t[0]){     //一個目がとれない場合
    strcpy(JDG,"NG");
    return 0+1;
  }
  else{              //一個目は取れる場合
    //初期条件
    DP[0][0]=p[0];
    min = p[0];

    for(i=1;i<n;i++){
      min = INF;
      for(j=0;j<BalMax;j++){
        DP[j][i] = val(j,i);
        if(DP[j][i]<min)min = DP[j][i];
      }
      if(min >= INF){        //とれなかった
        strcpy(JDG,"NG");
        break;
      }
    }

    if(i==n){//全部取れた場合（for ループを break で抜けない）
      strcpy(JDG,"OK");
      return min+p[n-1];
    }else{   //for ループを break で抜けた
      return i+1;
    }
  }
}

int main(void){
  int i;

  while(1){
```

```
    scanf(" %d ", &n);
    if(!n)break;
    for(i=0;i<n;i++)scanf("%d %d",&p[i],&t[i]);

    initDP();   //初期化

    printf("%s %d\n",JDG ,solve());
  }

  return 0;
}
```

6 ライブラリ

以下では独特の入力に対応した入出力、C 言語では自力で書かなければならないリスト、スタック、キュー、そして数論で使うエラトステネスの篩を関数化したものを紹介する。

6.1 入力

6.1.1 通常の入力

- input

  n

  s_1

  s_2

  s_3

  $\cdots$

  s_n

- Sample Input

```
3
1000
342
0
5
2
2
9
11
932
5
```

```
300
1000
0
200
400
8
353
242
402
274
283
132
402
523
0
```

- テンプレート

```
int main(void){
  int i;
  int n;
  int x;

  while(1){
    scanf("%d",&n);
    if(!n)break;

    for(i=0;i<n;i++)scanf("%d",&x);
    ......
  }
  return 0;
}
```

6.1.2 マップ構造

- input

$c_{1,1}c_{1,2}\ldots c_{1,w}$

$c_{2,1}c_{2,2}\ldots c_{2,w}$

$\ldots$

$c_{h,1}c_{h,2}\ldots c_{h,w}$

- Sample Input

```
1 1
0
2 2
0 1
1 0
3 2
1 1 1
1 1 1
5 4
1 0 1 0 0
1 0 0 0 0
1 0 1 0 1
1 0 0 1 0
5 4
1 1 1 0 1
1 0 1 0 1
1 0 1 0 1
1 0 1 1 1
5 5
1 0 1 0 1
0 0 0 0 0
1 0 1 0 1
0 0 0 0 0
1 0 1 0 1
0 0
```

- テンプレート

```
int w,h;
int map[100][100];

int main(void){
  int i,j;

  while(1){
    scanf(" %d %d ",&w,&h);
    if(!w && !h)break;

    for(i=0;i<h;i++){
      for(j=0;j<w;j++){
        scanf(" %d  ",&map[i][j]);
      }
    }
    ......
```

```
  }
  return 0;
}
```

6.1.3 文字列の入力

- input

  n
  $line_1$
  $line_2$
  ...
  $linen$

- Sample Input

```
3
nai tiruvantel ar varyuvantel i valar tielyama nu vilya
qua ist qda quang ncw psts
svampti tsuldya jay quadal ciszeriol
```

- テンプレート

```
int main(void){
  int i;
  int n;
  int str[256];

  while(1){
    scanf(" %d ",&n);
    if(!n)break;

    for(i=0;i<n;i++)fgets(str,256,stdin);
    ……
  }
  return 0;
}
```

6.2 単方向リスト

リファレンス

List *list_add_head(List *l, int a) リスト l の先頭にデータ a を追加したものを返す.

List *list_add_tail(List *l, int a) リスト l の末尾にデータ a を追加したものを返す.

List *list_add_at(List *l, int a, int i) リスト l の i 番目にデータ a を挿入したものを返す.

List *list_delete_head(List *l) リスト l から先頭要素を削除したものを返す.

List *list_delete_tail(List *l) リスト l の末尾要素を削除したものを返す.

List *list_delete_at(List *l, int i) リスト l から i 番目の要素を削除したものを返す.

図 10: 関数 list_add_head

図 11: 関数 list_add_tail

図 12: 関数 list_add_at

図 13: 関数 list_delete_head

図 14: 関数 list_delete_tail

図 15: 関数 list_delete_at

ソースコード

```
#include <stdlib.h>

typedef struct _list{
  struct _list *next;
  int a;
}List;

List *list_add_head(List *l, int a){
  List *car;
  car = (List *)malloc(sizeof(List));
  car->a = a;
  car->next = l;
```

```
  return(car);
}

List *list_add_tail(List *l, int a){
  List *car;
  if(l == NULL){
    car = (List *)malloc(sizeof(List));
    car->a = a;
    car->next = NULL;
    return(car);
  }
  if(l->next == NULL){
    l->next = (List *)malloc(sizeof(List));
    l->next->a = a;
    l->next->next = NULL;
    return(l);
  }
  else list_add_tail(l->next, a);
  return(l);
}

List *list_add_at(List *l, int a, int i){
  List *car, *cdr;
  switch(i){
  case 0:
    car = (List *)malloc(sizeof(List));
    car->a = a;
    car->next = l;
    return(car);
    break;
  case 1:
    cdr = l->next;
    l->next = (List *)malloc(sizeof(List));
    l->a = a;
    l->next->next = cdr;
    return(l);
    break;
  default:
    list_add_at(l->next, a, i-1);
  }
  return(l);
}
```

```
List *list_delete_head(List *l){
  List *cdr;
  cdr = l->next;
  free(l);
  return(cdr);
}

List *list_delete_tail(List *l){
  if(l->next == NULL){
    free(l);
    return(NULL);
  }
  if(l->next->next == NULL){
    free(l->next);
    l->next = NULL;
  }
  else{
    list_delete_tail(l);
  }
  return(l);
}

List *list_delete_at(List *l){
  List *cdr;
  switch(i){
  case 0:
    cdr = l->next;
    free(l);
    return(cdr);
    break;
  case 1:
    cdr = l->next;
    l->next = l->next->next;
    break;
  default:
    list_delete_at(l->next, i-1);
  }
  return(l);
}

int list_get_at(List *l, int i){
  if(i <= 0) return(l->a);
  return(list_get_at(l->next, i-1));
```

```
}

int list_has_data(List *l, int a){
  if(l == NULL) return(0);
  if(l->a == a) return(1);
  return(list_has_data(l->next, a);
}

int list_size(List *l){
  if(l == NULL) return(0);
  return(1+list_size(l->next));
}

void list_free(List *l){
  if(l != NULL){
    list_free(l->next);
    free(l);
  }
}
```

6.3 スタック

リファレンス

#define STACK_MAX 1000 スタックに格納できるデータの最大数.

Stack *stack_new(void) 新たなスタックを生成する.

void stack_free(Stack *s) スタック s を解放する.

void stack_push(Stack *s, int x) スタック s にデータ x を push する.

int stack_pop(Stack *s) スタック s からデータを 1 つ pop する.

int stack_size(Stack *s) スタック s に積まれているデータの数を返す.

int stack_has_item(Stack *s, int x) スタック s にデータ x が入っていれば 1, 入っていなければ 0 を返す.

ソースコード

```
#include <stdlib.h>

#define STACK_MAX 1000

typedef struct{
```

図 16: 関数 stack_new

図 17: 関数 stack_free

図 18: 関数 stack_push

図 19: 関数 stack_pop

```
  int size;
  int x[STACK_MAX];
}Stack;

Stack *stack_new(void){
  Stack *s;
  s = (Stack *)malloc(sizeof(Stack));
  s->size = 0;
  return(s);
}

void stack_free(Stack *s){
  free(s);
}

void stack_push(Stack *s, int x){
  if(s->size < STACK_MAX){
    s->x[(s->size)++] = x;
  }
}

int stack_pop(Stack *s){
  return((s->size > 0)?s->x[--(s->size)]:0);
}

int stack_size(Stack *s){
  return(s->size);
}

int stack_has_item(Stack *s, int x){
  int i;
```

```
  for(i=0; i<(s->size); i++){
    if(s->x[i] == x) return(1);
  }
  return(0);
}
```

6.4　キュー

リファレンス

#define QUEUE_MAX 1000 キューに格納できるデータの最大数.

Queue *queue_new(void) 新たなキューを作成する.

void queue_free(Queue *q) キュー q を解放する.

void queue_enqueue(Queue *q, int x) キュー q にデータ x を enqueue する.

int queue_dequeue(Queue *q) キュー q からデータを 1 つ dequeue する.

int queue_size(Queue *q) キュー q に入っているデータの数を返す.

int queue_has_item(Queue *q, int x) キュー q にデータ x が入っていれば 1, 入っていなければ 0 を返す.

図 20: 関数 queue_new

図 21: 関数 queue_free

図 22: 関数 queue_enqueue

図 23: 関数 queue_dequeue

ソースコード

```
#define QUEUE_MAX 1000

typedef struct{
  int x[QUEUE_MAX];
  int head, size;
}Queue;
```

```
Queue *queue_new(void){
  Queue *q;
  q = (Queue *)malloc(sizeof(Queue));
  q->head = 0;
  q->size = 0;
  return(q);
}

void queue_enqueue(Queue *q, int x){
  if(q->size < QUEUE_MAX){
    q->x[(q->head + q->size)%QUEUE_MAX] = x;
    q->size++;
  }
}

int queue_dequeue(Queue *q){
  int x=0;
  if(q->size > 0){
    x = q->x[q->head];
    q->head = (q->head + 1)%QUEUE_MAX;
    q->size--;
  }
  return(x);
}

void queue_free(Queue *q){
  free(q);
}

int queue_size(Queue *q){
  return(q->size);
}

int queue_has_item(Queue *q, int x){
  int i, index;
  for(i=0; i<(q->size); i++){
    index = (q->head + i)%QUEUE_MAX;
    if(q->x[index] == x) return(1);
  }
  return(0);
}
```

6.5 素数

リファレンス

void make_prime_flag_table(int n, int f[]) 0 からまでの素数表 f を作成する．f は n+1 個の領域を確保すること．

void make_prime_number_table(int n, int p[]) 0 から n までの素数配列 p を作成する．f は n+1 個の領域を確保すること．

int is_prime(int n) n が素数なら 1, そうでない場合は 0 を返す.

表 2: 素数表 f

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f_i	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

表 3: 素数配列 p

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
p_i	0	2	3	5	7	11	13	17	19	23	29	31	37	41	47	53

ソースコード

```
void make_prime_flag_table(int n, int a[]){
  int i, j;
  a[0] = a[1] = 0;
  a[2] = 1;
  for(i=3; i<=n; i++){
    a[i] = i&1;
  }
  i=3;
  while(i*i <= n){
    for(j=i<<1; j<=n; j+=i){
      a[j] = 0;
    }
    do{i+=2;}while(!a[i]);
  }
}

void make_prime_number_table(int n, int a[]){
  int p, i, j;
  a[0] = 0;
  a[1] = 2;
  i = 2;
```

```
  p = 3;
  while(i<=n){
    for(j=1; j<=i; j++){
      if(j==i) a[i++] = p;
      if(!(p%a[j]) break;
    }
    p += 2;
  }
}

int is_prime(int n){
  int i;
  if(n <= 1) return(0);
  if(n == 2) return(1);
  if(!(n & 1)) return(0);
  for(i=3; i*i<=n; i+=2){
    if(!(n%i)) return(0);
  }
  return(1);
}
```

7 参考文献・Web ページ

- 目指せ！プログラミング世界一ー大学対抗プログラミングコンテスト ICPC への挑戦
  筧 捷彦 著　近代科学社
- プログラミングコンテストチャレンジブック
  秋葉 拓哉、岩田 陽一、北川 宜稔 著　毎日コミュニケーションズ
- MAYAH.JP ACM-ICPC 国内予選・アジア地区予選・世界大会
  http://cm.baylor.edu/welcome.icpc
- ACM-Japan
  http://www.acm-japan.org/
- ACM/ICPC 国内予選突破の手引き
  http://www.deqnotes.net/acmicpc/
- ACM/ICPC OB/OG の会
  http://acm-icpc.aitea.net/index.php?FrontPage
- 秋田大学 ICPC 対策室@ wiki
  http://www23.atwiki.jp/akitaicpc/pages/1.html
- ACM/ICPC プログラミングコンテストに備えて
  http://ww.infor.kanazawa-it.ac.jp/ koblab/home/e1504310/acm/index.html
- Aizu Online Judge
- PKU