

Competitive Programmer's Handbook

日本語訳

Antti Laaksonen

和訳: kmjp

Draft December 22, 2017

Contents

序文	v
I 基礎テクニック	1
1 イントロダクション	3
1.1 プログラミング言語	3
1.2 入出力	4
1.3 数の扱い	6
1.4 コードの短縮	8
1.5 数学	10
1.6 コンテストと情報源	14
2 時間計算量	17
2.1 計算手順	17
2.2 計算量クラス	20
2.3 実行効率の見積もり	21
2.4 最大部分列和	21
3 ソート	25
3.1 ソートの理論	25
3.2 C++ におけるソート	29
3.3 二分探索	31
4 データ構造	35
4.1 動的配列	35
4.2 集合型	37
4.3 連想配列型	38
4.4 イテレータと区間	39
4.5 その他のデータ構造	41
4.6 ソートとの比較	44
5 全探索	47
5.1 部分集合の生成	47
5.2 順列の生成	49
5.3 バックトラック法	50
5.4 探索の枝刈り	51
5.5 半分全列挙	54

6	貪欲アルゴリズム	57
6.1	コインの問題	57
6.2	スケジューリング	58
6.3	タスクと締切	60
6.4	和の最小化	61
6.5	データ圧縮	62
7	動的計画法	65
7.1	コイン問題	65
7.2	最長増加部分列	70
7.3	グリッド中のパス	71
7.4	ナップサック問題	72
7.5	編集距離	74
7.6	敷き詰め方の数え上げ	75
8	ならし解析	77
8.1	尺取り法	77
8.2	nearest smaller elements	79
8.3	スライド最小値	80
	Bibliography	83

序文

この本の目的は、読者に競技プログラミングの徹底的な入門書となることである。前提として、読者はプログラミングの基礎知識を持つものとするが、競技プログラミングのバックグラウンドは必要としない。

この本は特にアルゴリズムを学び国際情報オリンピック (IOI) や ACM 国際大学対抗プログラミングコンテスト (ICPC) への参加したいと考えている学生向けである。もちろん競技プログラミングに興味があれば誰にでも適している。

良い競技プログラマになるには長い時間を要するが、それは多くの学びの機会でもある。読者はこの本を読み、問題を解き、コンテストに出る時間を通じてアルゴリズムをより良く理解できるようになるだろう。

本書は継続的に進展している最中である。この本へのフィードバックは `ahslaaks@cs.helsinki.fi` まで送ってほしい。

Helsinki, October 2017
Antti Laaksonen

Part I

基礎テクニック

Chapter 1

イントロダクション

競技プログラミングは2つのトピック、すなわちアルゴリズムの設計と実装からなる。

アルゴリズムの設計は問題の解決と数学的な思考からなる。それには問題を分析し、創造的に解くスキルが求められる。問題を解くアルゴリズムとは、正確かつ効率的でなければならない。そして問題の中核は、しばしば効率的なアルゴリズムを考案することとなる。

もちろん競技プログラマにはアルゴリズムの理論的な知識は重要である。一般的に問題の解法とは、よく知られたテクニックと、新たな洞察の組み合わせである。競技プログラミングで現れる様々なテクニックは、アルゴリズムの科学的な研究における基礎を形作るものともなるだろう。

アルゴリズムの実装はプログラミングスキルを要求する。競技プログラミングでは、解答は実装したアルゴリズムをテストケースを使ってテストし、評価される。そのため、アルゴリズムのアイデアが正確であるだけでなく、実装も正確でなければならない。

コンテストにおけるよいコーディングスタイルとは、複雑でなく明快であることだ。時間が限られているため、プログラムは素早く書かれなければならない。一般的なソフトウェア開発と異なり、コードは短く（通常多くて数百行）で、コンテスト後は継続的なメンテナンスは不要である。

1.1 プログラミング言語

現在、コンテストで広く使われる言語はC++、Python、Javaである。例えばGoogle Code Jam 2017の上位3000人を見ると、79%がC++、16%がPython、8%がJavaを使用している。一部の参加者は複数の言語を使い分けていた。

多くの人は競技プログラミングにはC++が最適な選択と考える。実際C++はほぼすべてのコンテストで利用できる。C++の利点は、効率的であることと標準ライブラリが充実（特にデータ構造とアルゴリズム）している点である。

一方、複数の言語を使いこなし、長短を理解できるようになることも重要である。例えば大きな整数を扱う必要がある場合、ビルトインで多倍長整数演算をサポートしているPythonはよい選択肢である。ただし、多くの問題は特定言語が不公平に有利になるようには設定されない。

本書のプログラムはすべて C++ で書かれており、標準ライブラリの機能を用いる。C++ のバージョンは、現在ほとんどのコンテストで利用可能である C++11 標準に従う。あなたがまだ C++ によるプログラミングができないなら、今こそ勉強するよい機会だ。

C++ コードテンプレート

競技プログラミングにおける C++ の典型的なコードテンプレートは以下のようになる:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // 解答はここ
}
```

先頭行の `#include` は g++ コンパイラに標準ライブラリ全体を `include` させる指示である。これがないと、`iostream`, `vector`, `algorithm` といったライブラリを個別に `include` しなければならなくなってしまう。

`using` の行は、標準ライブラリのクラスや関数を直接的に利用可能とする宣言である。この行がないと `std::cout` と書く必要があるところを、後半の `cout` とだけ書けば済む。

このコードは以下のコマンドでコンパイルする:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

このコマンドはソースコード `test.cpp` からバイナリファイル `test` を生成する。コンパイラは C++11 標準に従い (`-std=c++11`)、コードを最適化し (`-O2`)、かつエラーにつながりかねない警告をすべて表示する (`-Wall`)。

1.2 入出力

ほとんどのコンテストでは、入出力に標準ストリームを用いる。C++ では、標準ストリームは入力用が `cin`、出力用が `cout` である。加えて C 言語の `scanf` や `printf` も利用できる。

プログラムの入力は一般的に空白や改行区切りの数値や文字列である。これらは `cin` から下記のように読み込める:

```
int a, b;
string x;
cin >> a >> b >> x;
```

このコードは、入力の要素ごとに 1 つ以上の空白または改行がある場合を前提とする。例えば上記コードは下記いずれの入力も読み込むことができる。

```
123 456 monkey
```

```
123    456  
monkey
```

cout は下記のように出力に用いられる:

```
int a = 123, b = 456;  
string x = "monkey";  
cout << a << " " << b << " " << x << "\n";
```

改行"\n" は endl より高速に動作する。endl は flush 処理を伴うためである。

C 言語の scanf および printf は C++ の標準ストリームの代替となる。これらは若干速いが、使いこなすのは難しい。以下のコードは標準入力から 2 つの整数を読み込む:

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

以下のコードは 2 つの整数を出力する:

```
int a = 123, b = 456;  
printf("%d %d\n", a, b);
```

しばしば空白の有無に関わらず行全体を読み込む場合がある。これは getline で解決できる。

```
string s;  
getline(cin, s);
```

データ量が事前に不明な場合、このようなループも有用である:

```
while (cin >> x) {  
    // code  
}
```

このループはこれ以上入力データがなくなるところまで、データを 1 つずつ読み込む。

一部コンテストでは、入出力にファイルを用いる。簡単な対策は、以下のコードを先頭に加えることである:

```
freopen("input.txt", "r", stdin);  
freopen("output.txt", "w", stdout);
```

そうすると、以後ファイル"input.txt" を入力、"output.txt" を出力とし、普通に標準ストリームを利用可能になる。

1.3 数の扱い

整数

競技プログラミングで最も使われる整数 (**integer**) 型は `int` 型である。この型は 32bit 長で、 $-2^{31} \dots 2^{31} - 1$ 、すなわち約 $-2 \times 10^9 \dots 2 \times 10^9$ の範囲の整数を表せる。`int` 型で不足するならば 64bit 型の `long long` を利用できる。こちらは $-2^{63} \dots 2^{63} - 1$ 、すなわち約 $-9 \times 10^{18} \dots 9 \times 10^{18}$ の範囲の整数を表せる。

以下のコードは `long long` 型変数を定義する:

```
long long x = 123456789123456789LL;
```

末尾の LL はこの定数が `long long` 型であることを示す。

`long long` 型を扱う場合に起こしがちな誤りとして、`int` 型と混在してしまうことがある。例えば以下のコードは想定外のバグを含んでいる:

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

変数 `b` は `long long` 型であるが、式 `a*a` 中の数はいずれも `int` 型であり、演算結果も同様に `int` 型となる。そのため変数 `b` は誤った結果を含む。この問題は、変数 `a` を `long long` 型にするか、式をキャストを用いて `(long long)a*a` に書き換えることで解決できる。

通常コンテストの問題は `long long` 型で足りるようにできている。しかしそれでも足りない場合に備え、`g++` コンパイラは 128bit 整数である `__int128_t` 型を提供していることを知っておくとよい。この型は $-2^{127} \dots 2^{127} - 1$ 、すなわち約 $-10^{38} \dots 10^{38}$ の範囲を表せる。しかし、この型はすべてのコンテストで利用可能ではない。(訳注:32bit `g++` を用いる **CodeForces** 等では利用できない)

モジュロ演算

$x \bmod m$ を x を m で割ったときの余りとして定義する。例えば $17 = 3 \times 5 + 2$ なので $17 \bmod 5 = 2$ である。

しばしば解が非常に大きな数値になる問題において、 m の剰余を答えるよう指示されることがある。(例: "10⁹+7 の剰余を出力せよ") これにより、実際の解はとても大きくても、プログラム中では `int` 型や `long long` 型の範囲で対応できる。

剰余の重要な性質として、和・差・積を剰余を取る前後どちらに行っても値が一致するというものがある。

$$\begin{aligned}(a+b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\(a-b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\(a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

そのため、我々はこれらの演算の前に毎回剰余を取ることで、数が大きくなり過ぎないようにすることができる。

例えば、下記コードは $n!$ (n の階乗) の m の剰余を求める:

```
long long x = 1;
for (int i = 2; i <= n; i++) {
    x = (x*i)%m;
}
cout << x%m << "\n";
```

通常我々は剰余が $0 \dots m-1$ の範囲に収まることを期待する。しかし、C++ 含め一部の言語では、負の値の剰余は 0 または負になってしまう。剰余が非負であることを確実にするためには、求めた剰余が負の場合 m を加えればよい:

```
x = x%m;
if (x < 0) x += m;
```

この処理は、コード中で減算を行うなど剰余が負になりうる場合でのみ行えばよい。

小数

競技プログラミングで用いられる浮動小数点数 (**floating point number**、以下単に小数と訳す) の型は 64-bit の `double` 型と、g++ コンパイラの独自拡張である 80-bit の `long double` 型である。ほとんどのケースは前者で十分だが、精度を高めるために後者を利用することもできる。

解の精度は問題文中で与えられるのが一般的である。解の出力には、`printf` で小数点以降の桁数を指定するのが簡単である。例えば以下のコードは小数 x を小数点以下第 9 位まで出力する:

```
printf("%.9f\n", x);
```

小数を扱う場合の難しさは、値によって正確に表現できず、真の値と格納した値で丸め誤差を生じることである。例えば以下のコードの出力は驚くべきものとなる:

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

丸め誤差のため、 x の値は真の計算結果 1 よりもわずかに小さくなる。

計算過程でこのような丸め誤差が生じるため、2 つの小数を `==` 演算子で比較することは危険である。小数比較のより良い方法は、差の絶対値が十分に小さい値 ϵ 未満なら等しい、とみなすことである。

実際には小数值は以下のように比較できる ($\epsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {
    // a and b are equal
}
```

小数は不正確であるが、上限以下の整数については正確に保持できることは覚えておくとよい。例えば `double` 型は絶対値 2^{53} 未満の整数は正確な値を保持できる。

1.4 コードの短縮

プログラムをできるだけ素早く書くため、競技プログラミングにおいて短いコードは理想的である。そのため競技プログラマは、しばしば型名やその他の部分で短い別名を使う。

型名

`typedef` コマンドはデータ型に短い別名を与えることができる。例えば `long long` は長いため、`ll` を与えることができる:

```
typedef long long ll;
```

以後、以下のコードは:

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

以下のように短くできる:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

`typedef` はより複雑な型にも適用できる。例えば下記のコードは整数値の `vector` に `vi`、2つの整数の `pair` に `pi` という別名を与える:

```
typedef vector<int> vi;
typedef pair<int,int> pi;
```

マクロ

コードを短縮する別の手法にマクロがある。マクロはある文字列をコンパイル前に変換する機能である。C++ では、マクロは `#define` キーワードを用いて定義する。

例えば、以下のマクロを定義する:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

以後、以下のコードは:

```
v.push_back(make_pair(y1,x1));  
v.push_back(make_pair(y2,x2));  
int d = v[i].first+v[i].second;
```

このように短縮できる:

```
v.PB(MP(y1,x1));  
v.PB(MP(y2,x2));  
int d = v[i].F+v[i].S;
```

マクロはパラメータを取れるので、ループの短縮などに利用できる。例えば以下のマクロを定義すると:

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

以後以下のようなコードは:

```
for (int i = 1; i <= n; i++) {  
    search(i);  
}
```

このように短縮できる:

```
REP(i,1,n) {  
    search(i);  
}
```

マクロはしばしば原因特定が難しいバグの要因となる。例えば数値の二乗を返す以下のマクロを考える:

```
#define SQ(a) a*a
```

このマクロは想定通りには動かない場合がある。例えば下記コードは:

```
cout << SQ(3+3) << "\n";
```

このように変換される:

```
cout << 3+3*3+3 << "\n"; // 15
```

より良い定義方法は下記のとおりである:

```
#define SQ(a) (a)*(a)
```

すると以下のコードは:

```
cout << SQ(3+3) << "\n";
```

このように変換され、正しい値が得られる。

```
cout << (3+3)*(3+3) << "\n"; // 36
```

1.5 数学

数学は競技プログラミングにおける重要な素養であり、競技プログラミングで成功するための必須スキルである。この章では、本書で以後登場する数学の重要な概念や公式について論ずる。

総和に関する公式

正整数 k に対する

$$\sum_{x=1}^n x^k = 1^k + 2^k + 3^k + \dots + n^k,$$

の形の総和は、次数 $k+1$ の多項式の形の閉形式となる。

例えば

$$\sum_{x=1}^n x = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

であり

$$\sum_{x=1}^n x^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

となる。¹

等差数列 (**arithmetic progression**) とは連続する要素の差 (公差) が等しい数列である。例えば,

3, 7, 11, 15

は公差 4 の等差数列である。等差数列の総和は下記の式で計算できる:

$$\underbrace{a + \dots + b}_{n \text{ 要素}} = \frac{n(a+b)}{2}$$

a は初項、 b は最後の項、 n は要素数を示す。例えば

$$3 + 7 + 11 + 15 = \frac{4 \cdot (3 + 15)}{2} = 36.$$

この式は、 n 要素の平均値が $(a+b)/2$ であることに基づく。

等比数列 (**geometric progression**) とは、連続する要素の比 (公比) が等しい数列である。

3, 6, 12, 24

¹ これらの総和の式には、**Faulhaber's formula** と呼ばれる一般形式があるが上級者向けの内容なのでここでは割愛する。

は公比 2 の等比数列である。等比数列の総和は下記の式で計算できる:

$$a + ak + ak^2 + \dots + b = \frac{bk - a}{k - 1}$$

a は初項、 b は最後の項、 k は公比を示す。例えば

$$3 + 6 + 12 + 24 = \frac{24 \times 2 - 3}{2 - 1} = 45.$$

となる。

この式は以下のように導ける。まず以下を定義する:

$$S = a + ak + ak^2 + \dots + b.$$

両辺を k 倍すると以下のようになり:

$$kS = ak + ak^2 + ak^3 + \dots + bk,$$

以下の式が得られる:

$$kS - S = bk - a$$

等比数列のよくある値として、以下は覚えておくとよい。

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = 2^n - 1.$$

調和級数 (**harmonic sum**) は以下の形の総和である:

$$\sum_{x=1}^n \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}.$$

この調和級数の上界は $\log_2(n) + 1$ である。これは、各項 $1/k$ をにおける k を、 k を超えない k に最も近い 2 の累乗に置き換えることで確認できる。例えば $n = 6$ において、この調和級数は以下ようになる:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

この上界は $\log_2(n) + 1$ 個の部分に分けられ ($1, 2 \times 1/2, 4 \times 1/4$, 以下同様)、個々の総和は 1 以下である。 ,

集合論

集合 (**set**) とは要素のあつまりである。例えば集合

$$X = \{2, 4, 7\}$$

は要素 2, 4, 7 を含む。記号 \emptyset は空集合を意味する。 $|S|$ は集合 S の大きさ、すなわち要素数を意味する。上記の例でいえば $|X| = 3$ である。

集合 S が要素 x を含むことを、 $x \in S$ と表し、そうでない場合 $x \notin S$ と表す。例えば上記集合において

$$4 \in X \quad \text{かつ} \quad 5 \notin X$$

である。

集合に対する演算により、新たな集合を作成できる。

- 共通部分 (**intersection**) $A \cap B$ は、 A と B 両方に含まれる要素のみで構成された集合である。例えば、 $A = \{1, 2, 5\}$ で $B = \{2, 4\}$ のとき、 $A \cap B = \{2\}$ である。
- 和集合 (**union**) $A \cup B$ は、 A と B のいずれか片方または両方に含まれる要素のみで構成された集合である。例えば、 $A = \{3, 7\}$ で $B = \{2, 3, 8\}$ のとき、 $A \cup B = \{2, 3, 7, 8\}$ である。
- 補集合 (**complement**) \bar{A} は S に含まれない要素のみで構成された集合である。補集合の概念はすべてのありうる要素を含む全集合 (**universal set**) によって定まる。例えば、 $A = \{1, 2, 5, 7\}$ で全集合が $\{1, 2, \dots, 10\}$ のとき、 $\bar{A} = \{3, 4, 6, 8, 9, 10\}$ となる。
- 差 (**difference**) $A \setminus B = A \cap \bar{B}$ は A に含まれるが B に含まれない要素のみで構成された集合である。 B 自体は A に含まれない要素を含んでもよいことに注意すること。例えば $A = \{2, 3, 7, 8\}$ で $B = \{3, 5, 8\}$ のとき、 $A \setminus B = \{2, 7\}$ である。

A の各要素が S にも含まれている場合、 A を S の部分集合 (**subset**) と呼ぶ。集合 S に対して、空集合を含め $2^{|S|}$ 通りの部分集合が存在する。例えば集合 $\{2, 4, 7\}$ の部分集合は下記のとおりである：

$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}$ および $\{2, 4, 7\}$ 。

しばしば使われる集合として、 \mathbb{N} (自然数), \mathbb{Z} (整数), \mathbb{Q} (有理数), \mathbb{R} (実数) がある。 \mathbb{N} は $\mathbb{N} = \{0, 1, 2, \dots\}$ と $\mathbb{N} = \{1, 2, 3, \dots\}$ の 2 通りの定義があり、状況によって使い分けられる。

集合の構成法として、関数 $f(n)$ を用いた内包形式もある：

$$\{f(n) : n \in S\},$$

これは、 S の各要素 n に対し $f(n)$ で構成される要素ならなる集合である。例えば以下の集合はすべての偶数を含む：

$$X = \{2n : n \in \mathbb{Z}\}$$

論理演算

論理演算で取りうる値は真 (**true**, 1) か偽 (**false**, 0) である。特に重要な演算子として、 \neg (否定, **negation**), \wedge (論理積, **conjunction**), \vee (論理和, **disjunction**), \Rightarrow (含意, **implication**), \Leftrightarrow (同値, **equivalence**) がある。これらの演算の意味は下記の表のとおりである。

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

式 $\neg A$ は、 A と反対の値を返す。式 $A \wedge B$ は、 A と B 両方が真の場合のみ真である。式 $A \vee B$ は、 A と B いずれかもしくは両方が真の場合に真である。式 $A \Rightarrow B$ は、 A が真の場合に B も真であれば真である。式 $A \Leftrightarrow B$ は、 A と B が両方とも真または両方とも偽のとき真である。

述語 (**predicate**) は引数に応じて真偽いずれかの値を返す式である。述語は通常大文字で表現される。例えば $P(x)$ を x が素数であるときに真を返す述語とする。この定義に沿えば、 $P(7)$ は真で $P(8)$ は偽である。

量子化子 (もしくは限量子、限定子、**quantifier**) とは、集合の要素と式を結びつける役割を持つ。最も重要な量子化子は \forall (全称量子化子, **for all**) と \exists (限定量子化子, **there is**). である。例えば式

$$\forall x(\exists y(y < x))$$

は、集合のあらゆる要素 x に対して、 x より小さな y となるような要素 y が存在することを示す。これは整数集合に対しては真であるが、自然数集合に対しては偽である。

ここまで挙げた表記法により、さまざまな条件記述が可能となる。例えば

$$\forall x((x > 1 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(a > 1 \wedge b > 1 \wedge x = ab))))$$

は x が 1 より大きい合成数であれば、1 より大きな 2 つの数値 a, b のうち積が x となるものが存在することを意味する。この条件は整数集合について真である。

関数

関数 $\lfloor x \rfloor$ は実数 x からそれを超えない最大の整数に変換し、関数 $\lceil x \rceil$ は実数 x からそれ以上の最小の整数に変換する。例えば下記の通り:

$$\lfloor 3/2 \rfloor = 1 \quad \text{および} \quad \lceil 3/2 \rceil = 2.$$

関数 $\min(x_1, x_2, \dots, x_n)$ や $\max(x_1, x_2, \dots, x_n)$ は値 x_1, x_2, \dots, x_n のうち最小もしくは最大値を取り出す。例えば下記の通り:

$$\min(1, 2, 3) = 1 \quad \text{および} \quad \max(1, 2, 3) = 3.$$

階乗 (**factorial**) $n!$ は下記の通り定義できる:

$$\prod_{x=1}^n x = 1 \times 2 \times 3 \times \dots \times n$$

もしくは再帰的な定義もできる:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

フィボナッチ数列 (**Fibonacci numbers**) は多くの事例で登場する。この数列は下記の通り再帰的に定義される:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

フィボナッチ数列の先頭は以下のような数列である:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

フィボナッチ数列を求める内包形式の公式もあり、**Binet's formula** と呼ばれる:

$$f(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}.$$

対数

数 x の対数 (**logarithm**) は式 $\log_k(x)$ で現れる。 k は対数の底である。定義より、 $\log_k(x) = a$ となるのは $k^a = x$ の場合である

対数の重要な特徴として、 $\log_k(x)$ は x が 1 に到達するまで k で割っていったときの除算の回数に一致する点がある。

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

対数はしばしばアルゴリズムの分析に使われる。それは、多くの効率的なアルゴリズムが処理対象を 1 ステップで半分にしていくものであるためである。そのため我々はアルゴリズムの効率を入力データの対数値から見積もることがある。

数 n の自然対数 (**natural logarithm**) $\ln(x)$ とは、底に $e \approx 2.71828$ を用いたときの対数値である。

対数の別の用途としては、数 x を b 進数表記したときの桁数が $\lfloor \log_b(x) + 1 \rfloor$ となることである。例えば 123 の 2 進数表記は 1111011 であり、 $\lfloor \log_2(123) + 1 \rfloor = 7$ である。

1.6 コンテストと情報源

IOI

国際情報オリンピック (The International Olympiad in Informatics, IOI) は高校生を対象として毎年開催されるプログラミングコンテストである。各国 4 名からなるチームが参加する。例年 80 か国から計 300 名程度が参加する。

IOI は 5 時間のコンテスト 2 回からなる。参加者はそれぞれで様々な難易度の 3 問のタスクに取り組む。各問題はサブタスクがあり、回答状況によって部分点が得られる。参加者は国ごとにチームを構成するものの、競技は個人で行う。

IOI シラバス [41] は IOI のレギュレーションについて明記している。IOI シラバスのほとんどの内容はこの本で網羅されている。

IOI の参加者は国内コンテストを通じ選抜される。IOI の前、各地の地域別コンテストが行われる。例えば the Baltic Olympiad in Informatics (BOI), the Central European Olympiad in Informatics (CEOI), the Asia-Pacific Informatics Olympiad (APIO) がある。

一部の国では将来の IOI 選手に向け練習用のオンラインコンテストを実施している。例えば Croatian Open Competition in Informatics [11] や USA

Computing Olympiad [68] がある。加えて、大量の問題セットからなるポーランドのコンテストサイトも利用できる [60]。

ICPC

国際大学対抗プログラミングコンテスト (The International Collegiate Programming Contest, ICPC) は大学生を対象として毎年開催されるコンテストである。各チームは 3 名の学生で構成される。IOI と異なり、こちらは 3 名で 1 台のコンピュータを用いて一緒に競技に参加する。

ICPC は何段階かのステージを経て、最終的に成績の優秀なチームのみ決勝に招待される。全参加者は万単位であるが、決勝に残るのはわずかであり、² 地域によっては決勝に残るだけでも偉大な成績であると言える。

ICPC では 5 時間で約 10 問の問題に取り組む。各問題への採点には部分点はなく、全テストケースで正答しなければ正解とみなされない。コンテスト中、参加者は他チームの状況を知ることができる。しかし最後の 1 時間はスコア表は凍結され、他の参加者には以後の状況が公開されなくなる。

ICPC の出題範囲は IOI ほど明確には規定されていない。ただし、IOI より多くの知識と数学のスキルを要することは確かである。

オンラインコンテスト

誰でも参加できるオンラインコンテストも多数ある。現在最も活発なのは Codeforces で、毎週コンテストが開催されている。Codeforces は参加者を 2 つの division に分けており、初級者は Div2、上級者は Div1 で競う。他のコンテストサイトには、AtCoder、CS Academy、HackerRank、Topcoder などがある。

一部企業はオンサイト決勝を含むオンラインコンテストを行っている。例として、Facebook Hacker Cup, Google Code Jam, Yandex.Algorithm が挙げられる。これら企業はコンテストを採用の一環として利用している。そのためコンテストでよい結果を示すことは、自身のスキルを証明するよい方法とも言える。

書籍

競技プログラミングやアルゴリズムを用いた問題解決について、(もちろん本書も含めて) すでにいくつかの本が存在する:

- S. S. Skiena and M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]
- S. Halim and F. Halim: *Competitive Programming 3: The New Lower Bound of Programming Contests* [33]
- K. Diks et al.: *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions* [15]

²年によって変化はあるが、2017 年は 133 チームである。

上 2 つは初心者向けで、3 つ目は上級者向けの内容を含む。
もちろん一般的なアルゴリズムの解説書も有用である。著名な本は下記の通り:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein: *Introduction to Algorithms* [13]
- J. Kleinberg and É. Tardos: *Algorithm Design* [45]
- S. S. Skiena: *The Algorithm Design Manual* [58]

Chapter 2

時間計算量

競技プログラミングにおいてアルゴリズムの効率性は重要である。一般的に、遅いアルゴリズムの設計は簡単であるが、高速なアルゴリズムの発明は難しいチャレンジと言える。もしアルゴリズムが遅すぎれば、コンテストでは0点かせいぜい部分点しか得られない。

アルゴリズムの時間計算量 (**time complexity**) は、入力に対しどの程度の時間がかかるかを概算するための概念である。時間計算量のアイデアは、入力長をパラメータとする関数で実行時間を表すものである。この時間計算量を算出することで、アルゴリズムに対し実際に実装することなく十分に高速かどうか予測できる。

2.1 計算手順

アルゴリズムの時間計算量は $O(\dots)$ と表記される (3点ドットは何らかの関数である)。通常、変数 n で入力長を表す。例えば入力が整数列の場合 n は数列の要素数であり、入力が文字列ならば n は文字列長である。

ループ

アルゴリズムが遅い一般的な理由はループが深いことにある。深い多重ループを含むアルゴリズムほど遅くなる。もしアルゴリズムが k 重ループを含むとき、その時間計算量は $O(n^k)$ と表記される。

例えば、以下の一重ループの時間計算量は $O(n)$ である:

```
for (int i = 1; i <= n; i++) {  
    // code  
}
```

以下の二重ループでは時間計算量は $O(n^2)$ となる:

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        // code  
    }  
}
```

時間計算量の表記は、ループの正確な回数を示すものではなく、大よそのオーダーを示すものに過ぎない。下記のコードのループ回数はそれぞれ $3n$, $n+5$, $\lceil n/2 \rceil$ 回だが、いずれも時間計算量は $O(n)$ と表現される。

```
for (int i = 1; i <= 3*n; i++) {  
    // code  
}
```

```
for (int i = 1; i <= n+5; i++) {  
    // code  
}
```

```
for (int i = 1; i <= n; i += 2) {  
    // code  
}
```

別の例として、以下のコードの時間計算量は $O(n^2)$ である:

```
for (int i = 1; i <= n; i++) {  
    for (int j = i+1; j <= n; j++) {  
        // code  
    }  
}
```

手順

もしアルゴリズムがいくつかの手順を踏む場合、全体の時間計算量はここの手順の最大値となる。その理由は、最も大きな時間計算量の手順が通常コードのボトルネックとなるためである。

例えば以下のコードは 3 つの手順からなり、それぞれの時間計算量は $O(n)$, $O(n^2)$, $O(n)$ である。よって全体の時間計算量は $O(n^2)$ となる。

```
for (int i = 1; i <= n; i++) {  
    // code  
}  
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        // code  
    }  
}  
for (int i = 1; i <= n; i++) {  
    // code  
}
```


複数変数がある場合

時間計算量は複数パラメータに依存することもある。この場合、時間計算量の式も複数の変数を含む。

例えば以下のコードの時間計算量は $O(nm)$ である：

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= m; j++) {  
        // code  
    }  
}
```

再帰

再帰関数の時間計算量は、関数の実行回数と 1 回の実行あたりの時間計算量の積となる。

例えば以下の関数を考えてみる：

```
void f(int n) {  
    if (n == 1) return;  
    f(n-1);  
}
```

$f(n)$ は n 回の関数呼び出しを行い、個々の関数呼び出しにおける時間計算量は $O(1)$ である。よって全体の時間計算量は $O(n)$ となる。

別の例を考えてみよう：

```
void g(int n) {  
    if (n == 1) return;  
    g(n-1);  
    g(n-1);  
}
```

この例では、関数は $n = 1$ の場合を除けば 2 回関数呼び出しを行う。 g がパラメータ n を伴って実行されたときの挙動を見てみよう。下の表は 1 回の呼び出しに伴う関数呼び出し回数を示す：

呼び出し形式	実行回数
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	2^{n-1}

この表より、この関数の時間計算量は次式の通りとなる。

$$1 + 2 + 4 + \dots + 2^{n-1} = 2^n - 1 = O(2^n).$$

2.2 計算量クラス

このリストは、典型的な時間計算量を示している:

- $O(1)$ 定数時間 (**constant-time**) アルゴリズムの実行時間は入力サイズによらない。典型的な定数時間のアルゴリズムは、解を直接求める計算式である。
- $O(\log n)$ 典型的な対数時間 (**logarithmic**) アルゴリズムは、アルゴリズムを1ステップ進めるたびに入力サイズを半分にしていけるものである。そのようなアルゴリズムの実行時間は対数時間となる。なぜなら $\log_2 n$ は n が1になるまで半減させていく回数と一致するためである。
- $O(\sqrt{n})$ 平方根時間 (**square root algorithm**) アルゴリズムは $O(\log n)$ より遅いが $O(n)$ より速い。平方根の特性として $\sqrt{n} = n/\sqrt{n}$ が成り立つ。よって平方根 \sqrt{n} はある意味では入力の中央値と言える。
- $O(n)$ 線形時間 (**linear**) アルゴリズムは、入力を定数回辿るようなアルゴリズムである。これはしばしばありうる最善のアルゴリズムである。なぜなら、解を出力する前に通常は全入力データを一旦読み込む時点で、 $O(n)$ となるためである。
- $O(n \log n)$ この時間計算量は、しばしば入力のソートを意味する。というのも、効率のよいソートアルゴリズムの時間計算量がこのクラスであるためである。別の可能性として、アルゴリズム中である処理に $O(\log n)$ かかるようなものである場合もある。
- $O(n^2)$ 平方時間 (**quadratic**) アルゴリズムはしばしば二重ループを含む。例えば入力におけるペアを全探索する場合、 $O(n^2)$ となる。
- $O(n^3)$ 立方時間 (**cubic**) アルゴリズムはしばしば三重ループを含む。例えば入力における三つ組を全探索する場合、 $O(n^3)$ となる。
- $O(2^n)$ この時間計算量は、しばしば入力における全部分集合を探索する場合に現れる。例えば集合 $\{1, 2, 3\}$ の部分集合は $\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$ である。
- $O(n!)$ この時間計算量は、しばしば入力の並べ替え方を全探索する場合に現れる。例えば数列 $\{1, 2, 3\}$ を並べ替えたものには $(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$ がある。

時間計算量が定数 k を用いて最大で $O(n^k)$ の形で表されるとき、そのアルゴリズムは多項式時間 (**polynomial**) アルゴリズムである。上記計算量クラスのうち $O(2^n)$ と $O(n!)$ 以外は多項式時間アルゴリズムである。実用上は定数 k は小さいことが多く、またそのため多項式時間アルゴリズムは大よそ効率的とみなされることが多い。

この本のアルゴリズムのほとんどは多項式時間アルゴリズムである。しかし、世の中には誰も多項式時間の効率的な解法を持たない種々の重要な問

題がある。NP 困難 (**NP-hard**) な問題はそのような多項式時間アルゴリズムが知られていない問題クラスである。¹

2.3 実行効率の見積もり

アルゴリズムの時間計算量を求めることで、アルゴリズムを実装する前に問題を解くのに十分な効率かチェックすることができる。まず前提として、現在の計算機は秒間数億回の演算ができることを頭に入れておこう。

例えばプログラムの実行時間の上限が 1 秒で、入力サイズが $n = 10^5$ だとする。もし時間計算量が $O(n^2)$ のアルゴリズムを実行する場合、大体 $(10^5)^2 = 10^{10}$ 回の演算が実行される。この回数の演算は数十秒かかるため、このアルゴリズムは遅すぎると推測できる。

一方、入力サイズをもとに必要なアルゴリズムの時間計算量を予測することもできる。以下の表は実行時間 1 秒間で処理できるデータ量の概算値を示す。

入力サイズ	必要な時間計算量
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n がさらに大きい	$O(1)$ or $O(\log n)$

例えば入力サイズが $n = 10^5$ であれば、要求されるアルゴリズムの時間計算量はおそらく $O(n)$ か $O(n \log n)$ と推測できる。この情報は、明らかに間に合わない時間計算量のアルゴリズムの可能性を事前に排除できるため、アルゴリズムの設計を容易にする。

時間計算量はあくまで効率の概算であることという理解は重要である。なぜなら時間計算量は定数倍要因を省略しているためである。例えば時間計算量が $O(n)$ のアルゴリズムと言っても、実際の演算回数は $n/2$ かもしれないし $5n$ かもしれない。そしてその違いが実際の実行時間に与える影響は大きい。

2.4 最大部分列和

しばしば問題を解くうえで様々な時間計算量のアルゴリズムが考えられることがある。ここでは簡潔な $O(n^3)$ 解法のある古典的な問題を取り上げる。しかしより良いアルゴリズム設計で、時間計算量を $O(n^2)$ や $O(n)$ まで減らせることがわかっている。

n 要素の数列に対し、最大部分列和 (**maximum subarray sum**) を求めることを考える。最大部分列和とは、数列中のある連続した値の和として取りえる最大値である。² この問題は、数列中に負の値があるときに興味深いものとなる。例えば数列が以下の図のようになっているとする：

¹この話題に関する古典的な書籍には M. R. Garey's and D. S. Johnson's *Computers and Intractability: A Guide to the Theory of NP-Completeness* [28] がある。

²J. Bentley の書籍 *Programming Pearls* [8] でこの問題が有名になった。

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

the following subarray produces the maximum sum 10:

以下の部分列が最大部分列和 10 を成す。

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

ここでは空の部分列も許可される。そのため、最大の部分列の和は少なくとも 0 である。

アルゴリズム 1

わかりやすい解法は、部分列を全探索列挙し、それぞれの総和を求め最大値を保持することである。このアルゴリズムを実行するとこのようになる:

```
int best = 0;
for (int a = 0; a < n; a++) {
    for (int b = a; b < n; b++) {
        int sum = 0;
        for (int k = a; k <= b; k++) {
            sum += array[k];
        }
        best = max(best, sum);
    }
}
cout << best << "\n";
```

変数 a と b は部分列の先頭と末尾の位置を保持する。そしてその和が変数 sum に格納される。変数 $best$ は探索中に見つけた和の最大値を保持する。

このアルゴリズムは入力に対し三重ネストのループを含んでおり、時間計算量は $O(n^3)$ である。

アルゴリズム 2

アルゴリズム 1 の効率を改善する簡単な手法はループを取り除くことである。部分列の末尾を動かす際、同時に部分列の和を計算していけばループを 1 つ減らすことができる。その時のコードはこのようになる:

```
int best = 0;
for (int a = 0; a < n; a++) {
    int sum = 0;
    for (int b = a; b < n; b++) {
        sum += array[b];
        best = max(best, sum);
    }
}
cout << best << "\n";
```

変更後の時間計算量は $O(n^2)$ である。

アルゴリズム 3

驚くべきことに、この問題は $O(n)$ 時間³、すなわち単一ループのアルゴリズムで解くことができる。その基本的なアイデアは、部分列の末尾を探索していく過程で、その末尾に対応した最大部分和を適宜求めていくことである。

部分列の末尾の位置 k が固定されている場合に部分和が最大となる部分列を見つけるという部分問題を考えよう。そのような部分列には 2 つの可能性がある:

1. 部分列は k 番目の要素のみ含む。
2. 部分列は $k-1$ 番目の要素を末尾とする部分列に k 番目の要素を加えたものである。

後者のケースで考えると、我々は部分列の最大和を求めたいので、 $k-1$ 番目を末尾とする部分列としては、そのような部分列のうち最大和となるものを用いたい。よって、この問題は末尾の位置を左から右に動かしつつ最大部分列和を適宜求めていくことで効率的に解くことができる。

このアルゴリズムを実装するとこのようになる:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k], sum+array[k]);
    best = max(best, sum);
}
cout << best << "\n";
```

このアルゴリズムは入力データを探索する単一ループしか持たないため、時間計算量は $O(n)$ である。これは考えうる最良の時間計算量である。なぜならこの問題を解くうえでいかなるアルゴリズムでも各要素を最低 1 回は辿らないといけないためである。

効率の比較

実際これらのアルゴリズムがどの程度実用的か興味深いことだろう。下記の表は異なる入力サイズに対する実行時間を示している。

各実験において、入力ランダムに生成された。また入力を読み込む時間は含んでいない。

入力サイズ n	アルゴリズム 1	アルゴリズム 2	アルゴリズム 3
10^2	0.0 s	0.0 s	0.0 s
10^3	0.1 s	0.0 s	0.0 s
10^4	> 10.0 s	0.1 s	0.0 s
10^5	> 10.0 s	5.3 s	0.0 s
10^6	> 10.0 s	> 10.0 s	0.0 s
10^7	> 10.0 s	> 10.0 s	0.0 s

³[8]においてこの定数時間アルゴリズムは J. B. Kadane によって紹介されたため、しばしば **Kadane's algorithm** と呼ばれる

この比較より、入力サイズが小さい場合はどのアルゴリズムも十分高速であるが、入力サイズが大きくなると大きな差が生じることがわかる。アルゴリズム 1 は $n = 10^4$ 、アルゴリズム 2 は $n = 10^5$ ごろから遅くなるが、アルゴリズム 3 は大きな入力サイズでも十分に対応できる。

Chapter 3

ソート

ソート (**Sorting**) はアルゴリズム設計における基本的な問題である。データがソート済みだと処理が容易になることはよくあることなので、多くの効率的なアルゴリズムがソートを内部に含んでいる。

例えば、「数列中に等しい 2 つの要素があるか？」という問題は、ソートにより容易に解くことができる。数列中に 2 つの同じ要素があるならば、ソート後にそれらは隣接するため容易に探し出すことができる。「数列中の最頻値は何か？」という問題も同様に容易に解けるだろう。

ソートには様々なアルゴリズムがあるため、さまざまなアルゴリズム設計のテクニックをどう適用するか考えるのによい教材となる。効率的な汎用のソートアルゴリズムの時間計算量は $O(n \log n)$ であり、ソートを内部で利用する多くのアルゴリズムも同様の時間計算量となる。

3.1 ソートの理論

ソートの基本的な問題はこのとおりである:

n 要素の数列が与えられたとき、その要素を昇順に並べ替えよ

例えば以下のような数列は:

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

ソート後以下のようになる:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

$O(n^2)$ アルゴリズム

数列を並べ替える単純なアルゴリズムは時間計算量 $O(n^2)$ となる。そのようなアルゴリズムは短く、通常二重ループで構成される。有名な時間計算量 $O(n^2)$ のソートアルゴリズムとしてバブルソート (**bubble sort**) がある。これは各要素が値に応じて泡のように数列中を動くことに由来する。

バブルソートは n ラウンドの処理で構成される。このアルゴリズムでは、1 回のラウンド毎に数列の要素を 1 周処理して回る。もし連続する要素が正しくない、すなわち降順に並んでいる場合、それらを交換する。実装はこのようになる:

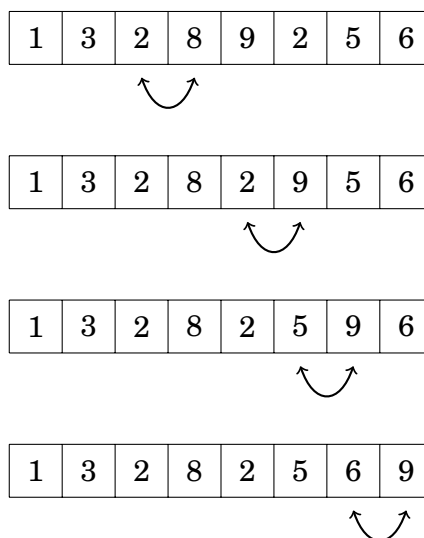
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j], array[j+1]);
        }
    }
}
```

初回のラウンドを終えると、最大の要素は正しい位置、つまり末尾に移動する。一般的に k ラウンド後には上位 k 番目までに大きい要素が正しい位置に移動する。よって n ラウンド後は全要素が正しい位置に移動したことになり、数列全体のソートが完了したことになる。

例えば以下の数列で:

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

バブルソートの最初のラウンドでは、要素は以下のように交換される:



転倒

バブルソートは、常に数列中で連続した要素のみを交換するアルゴリズムの一例である。そのようなアルゴリズムの時間計算量は常に少なくとも $O(n^2)$ となる。それは最悪ケースではソートに $O(n^2)$ 回の交換を要するためである。

ソートアルゴリズムの分析に役立つ概念に転倒 (**inversion**) がある。転倒とは $a < b$ でありながら $\text{array}[a] > \text{array}[b]$ となるような要素の対 ($\text{array}[a], \text{array}[b]$) のことである。つまり、ソート後にあるべき順番に対し誤った順番に並んでいる要素対のことである。例えば以下の数列:

1	2	2	6	3	5	9	8
---	---	---	---	---	---	---	---

は 3 つの転倒 (6,3), (6,5), (9,8) を含む。転倒数は数列をソートするための程度の作業が必要かを示す。転倒が一つもない状態がソート完了後の状態と言える。言い換えると、数列の要素がソート後の逆順に並んでいるとき、転倒数は最大値:

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}=O(n^2)$$

となる。

逆順にならんでいる連続する 2 要素の交換は、数列中から転倒を 1 つ減らす。よって、連続する要素のみを交換するソートアルゴリズムでは、1 回の交換で 1 つの転倒しか減らせないので、時間計算量は少なくとも $O(n^2)$ となる。

$O(n \log n)$ アルゴリズム

交換する要素を連続する要素に限らなければ、より効率よく時間計算量 $O(n \log n)$ で数列をソートできる。その一例がマージソート (**merge sort**¹) で、再帰処理を利用している。

数列の部分列 $\text{array}[a \dots b]$ のマージソートは、以下のように行う:

1. もし $a = b$ なら部分列はソート済みとみなせるので何もしない。
2. 中央となる位置 $k = \lfloor (a + b) / 2 \rfloor$ を求める。
3. 再帰的に前半部分 $\text{array}[a \dots k]$ をソートする。
4. 再帰的に後半部分 $\text{array}[k + 1 \dots b]$ をソートする。
5. 2 つのソート済み部分列 $\text{array}[a \dots k]$ と $\text{array}[k + 1 \dots b]$ をマージして 1 つのソート済み部分列 $\text{array}[a \dots b]$ を作る。

マージソートはステップ毎に対象とする部分列のサイズを半減させていくため、効率的なアルゴリズムである。再帰は $O(\log n)$ 段からなり、各段階の処理は時間計算量で $O(n)$ にかかる。部分列 $\text{array}[a \dots k]$ と $\text{array}[k + 1 \dots b]$ のマージは、両部分列がソート済みならば線形時間で終わる。

例として以下の数列のソートを考える:

1	3	6	2	8	2	5	9
---	---	---	---	---	---	---	---

この数列は以下のように 2 つの部分列に分割できる:

1	3	6	2
8	2	5	9

両部分列は、再帰的にこのようにソートされる:

1	2	3	6
2	5	8	9

最終的に、部分列はマージされて最終的にソート済み数列が得られる。

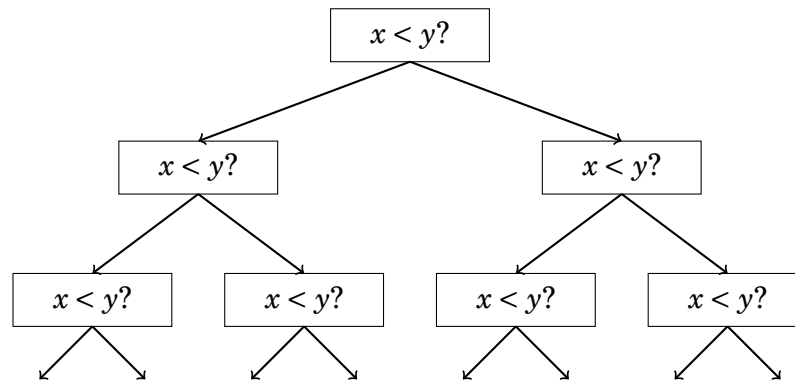
1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

¹[47] によると、マージソートは 1945 年にかの J. von Neumann によって考案された

ソートの下限

時間計算量が $O(n \log n)$ よりも小さいソートアルゴリズムは存在するだろうか？要素間の大小比較に基づくソートという前提のもとでは、不可能ということがわかっている。

時間計算量の下限は、ソートにおいて 2 要素の比較から得られる情報の量について考えると示すことができる。この手順はこのような木構造に落とすことができる：



ここで“ $x < y?$ ”はある 2 要素 x と y を比較することを示す。 $x < y$ であれば手続きは左側、そうでなければ右側に遷移し、移行手続きが続くものとする。この手続きの組み合わせは数列の並べ方の数だけあり得るので、全体で $n!$ 通り考えられる。この考えを進めると、木の高さは少なくとも

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n).$$

必要になる。

右辺の n 項のうち、後半 $n/2$ 項を $\log_2(n/2)$ に置き換えると、この式の下限がわかる。この式変形より

$$\log_2(n!) \geq (n/2) \times \log_2(n/2),$$

の関係がわかる。よって木の高さ、すなわち 2 要素の最小比較回数が最悪 $n \log n$ 回になることがわかる。

計数ソート

時間計算量の下限 $n \log n$ は、要素間の比較を用いないアルゴリズムには適用されない。その一例が計数ソート (**counting sort**) である。このアルゴリズムは数列中の要素が $0 \dots c$ の範囲の整数で、かつ $c = O(n)$ とすると時間計算量が $O(n)$ で済む。

このアルゴリズムでは、事前にカウンタ配列を作っておく。この配列のインデックスは元の数列の値に相当する。このアルゴリズムでは、元の数列の要素を順次見ていき、各値が何度現れるかを数え上げていく。

例えば、以下の数列に対し:

1	3	6	9	9	3	5	9
---	---	---	---	---	---	---	---

カウンタ配列はこのようになる:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

例えばカウンタ配列の3つめの位置には2が格納されている。これは元の数列で3が2回登場したことを意味する。

カウンタ配列の構築は $O(n)$ 時間がかかる。その後ソート後の配列の構築も、カウンタ配列の各要素を順次見ていくため $O(n)$ 時間がかかる。よって、計数ソートの時間計算量は全体で $O(n)$ となる。

計数ソートは非常に効率的だが、元の数列の値をカウンタ配列のインデックスに用いるため、 c が小さいときにしか利用できない。

3.2 C++ におけるソート

ソートアルゴリズムを今から自作することは良い考えではない。というのも、大抵のプログラミング言語ではよい実装がすでに利用可能であるためである。例えば C++ 標準ライブラリは、数列や他のデータ構造を容易にソートできる `sort` 関数を提供している。

このライブラリ関数を用いることには色々な利点がある。まず、自分で実装しなくてよいので時間を節約できる。また、ライブラリの実装は自作の関数ではそうそう超えられない程度まで正確かつ効率的に作られている。

この項では、C++ の `sort` 関数についてみていこう。以下のコードは `vector` を昇順にソートする:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

ソート後の `vector` の中身は以下のとおりである: [2,3,3,4,5,5,8]. デフォルトのソート順序は昇順である。しかしこのようにすれば逆順にできる:

```
sort(v.rbegin(),v.rend());
```

通常の配列は以下のようになる:

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a,a+n);
```

以下のコードは文字列 `s` をソートする:

```
string s = "monkey";
sort(s.begin(), s.end());
```

文字列をソートすると、文字列中の文字をソートしたものが得られる。例えば文字列 "monkey" をソートすると "ekmnoy" が得られる。

比較演算子

`sort` 関数は、ソート対象の配列の要素のデータ型が比較演算子 (**comparison operator**) を持つことを要求する。ソート過程で、この演算子は 2 つの要素のどちらを前にすべきか決定するのに用いる。

大抵の C++ のデータ型は組み込みの比較演算子を持つので、これらを要素とする配列は自動的にソート可能である。例えば数値型は値、文字列型はアルファベット順でソートされる。

ペア型 (pair) は、1 つめの要素 (first) でソートされる。しかし、もし 2 つのペアにおいて 1 つめの要素が等しい場合、続いて 2 つ目の要素 (second) によってソートされる:

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
```

上記 `vector` は、ソート後 (1,2), (1,5), (2,3) の順になる。

同様に、タプル型 (tuple) もまずは先頭要素でソートされる。先頭要素同士が等しい場合、次に 2 番目の要素、以下同様に処理される²

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
```

上記 `vector` はソート後 (1,5,3), (2,1,3), (2,1,4) の順となる。

ユーザー定義構造体

ユーザー定義の構造体には自動で比較演算子が定義されない。比較演算子は `operator<` として構造体内で定義されなければならない。この演算子は引数に同じ型のもう片方の要素をとり、呼び出し元要素が引数側の要素より小さいなら `true`、そうでなければ `false` を返すようにする。

²一部の古いコンパイラでは、タプルの生成に中括弧ではなく `make_tuple` 関数を要する。例えば {2,1,4} の代わりに `make_tuple(2,1,4)` としなければならない。

例えば頂点の x, y 座標を格納する P という型を作成したとする。この型の比較演算子として、例えばまず x 座標で大小比較し、 x 座標が等しいなら次に y 座標で大小比較するという定義が考えられる。

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x) return x < p.x;
        else return y < p.y;
    }
};
```

比較関数

`sort` 関数は、外部からコールバック関数として比較関数 (**comparison function**) を渡すことができる。例えば以下のような比較関数 `comp` を使うと、文字列をまず長さでソートし、同着なら次に辞書順で比較するというようなソートができる:

```
bool comp(string a, string b) {
    if (a.size() != b.size()) return a.size() < b.size();
    return a < b;
}
```

文字列型の `vector` はこのようにソートされる:

```
sort(v.begin(), v.end(), comp);
```

3.3 二分探索

数列中で要素を探す一般的な方法として、全要素を `for` ループで順次処理していく方法がある。例えばこのコードは数列中から要素 x を探す。

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x found at index i
    }
}
```

このアプローチは最悪ケースでは数列の全要素をチェックするので、時間計算量は $O(n)$ となる。数列の要素の並びに特徴がなく、他の追加情報がないという前提では、要素 x を探すのにこれは最良のアプローチである。

しかし、数列がソート済みである場合、事情が異なる。その場合、検索過程で要素の並びにより追加情報を得られるため、検索を高速化できる。以下の二分探索 (**binary search**) アルゴリズムは、ソート済み配列からデータを $O(\log n)$ で効率的に検索できる。

方法 1

二分探索を実装する一般的な方法は、辞書から単語を探す手順を真似ればよい。探索過程では、配列中で探索対象の領域を絞り込んでいく。この探索対象領域は初期状態では配列全体となるが、1 ステップごとに半分の大きさにしていく。

各ステップでは、探索対象領域の中央の要素をチェックする。もし中央の要素が検索値と一致するなら探索は終了する。そうでなければ、中央の要素と検索値の大小に応じて、探索対象を中央要素の手前か後いずれか半分として再帰的に探索を行う。

このアイデアを実装するとこのようになる:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // k番目の要素がxであることを見つけた
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

この実装では探索対象の領域は $a \dots b$ で表され、初期状態ではその領域は $0 \dots n-1$ である。このアルゴリズムはステップ毎に領域を半分にしていくので、時間計算量は $O(\log n)$ である。

方法 2

二分探索の別の実装法としては、効率的に配列を辿っていく手段が使える。アイデアとしては、最初は大きく要素間をジャンプしていき、検索値に近づくに従いジャンプの距離をだんだん短くしていくことになる。

この探索では、配列の先頭から末尾まで要素を辿っていく。初回のジャンプ距離は $n/2$ とする。ステップ毎にジャンプ距離は $n/4, n/8, n/16$ と半減していき、最終的には 1 となる。ジャンプの度に検索値が見つかるか、もしくは検索値を飛び越してしまいこれ以上あとには現れないかを判定していく。

このアイデアを実装するとこのようになる:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // k番目の要素がxであることを見つけた
}
```

この探索において、変数 b は現在のジャンプ距離を示す。このコード中、while ループは各ジャンプ距離に対し高々 2 回しか実行されないなので、このアルゴリズムの時間計算量は $O(\log n)$ である。

C++ の関数

C++ の標準ライブラリは、二分探索を用いて対数時間で探索を終える関数を持つ:

- `lower_bound` は配列中で値が少なくとも x である最小の要素へのイテレータを返す。
- `upper_bound` は配列中で値が x を超える最小の要素へのイテレータを返す。
- `equal_range` は上記両方のイテレータを返す。.

これらの関数は配列がソート済みであることを前提とする。条件を満たす要素が見つからない場合、これらの関数は配列の末尾へのイテレータを返す。例えば、このコードは配列中に x ちょうどの値を持つかどうかを示す。

```
auto k = lower_bound(array, array+n, x)-array;
if (k < n && array[k] == x) {
    // k番目の要素がxであることを見つけた
}
```

以下のコードは、配列中 x の出現回数を出力する:

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";
```

`equal_range` を使えばさらに短く書くことができる:

```
auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";
```

最小値の探索

二分探索の重要な応用として、ある関数の値が変化する位置の探索がある。例えば条件を満たす最小の値 k を求める問題を考えよう。関数 $ok(x)$ が与えられたとする。この関数は x が条件を満たすなら `true`、そうでなければ `false` を返す。ここでは、 $ok(x)$ は $x < k$ のとき `false`、 $x \geq k$ のとき `true` を返すようなものであることがわかっているとする。この状況をわかりやすく表すとこのようになる:

x	0	1	...	$k-1$	k	$k+1$...
$ok(x)$	false	false	...	false	true	true	...

この時、値 k は二分探索で求められる:

```

int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!ok(x+b)) x += b;
}
int k = x+1;

```

この探索では、 $ok(x)$ が false となる最大の x を求めている。そのため、次の値 $k = x + 1$ は $ok(k)$ が true となる最小の値となる。初期ジャンプ距離 z は、 $ok(z)$ が明らかに true となるのに十分な値にする帆必要がある。

このアルゴリズムは関数 ok を $O(\log z)$ 回呼び出すので、時間計算量は関数 ok に依存する。例えばこの関数が時間計算量 $O(n)$ であれば、この探索全体の時間計算量は $O(n \log z)$ となる。

最大値の探索

二分探索は、上に凸である（入力に応じて最初増加し、その後減少する）ような関数における最大値の探索にも応用可能である。以下の条件を満たす k を求める問題を考えよう：

- $x < k$ のとき $f(x) < f(x+1)$
- $x \geq k$ のとき $f(x) > f(x+1)$

アイデアとしては、 $f(x) < f(x+1)$ を満たす最大の x を二分探索で求めることである。そのような x に対し $f(x+1) > f(x+2)$ となるため $k = x + 1$ が求める値となる。このアイデアは以下のように実装できる：

```

int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;

```

通常の実二分探索と異なり、ここでは関数中で連続した入力値に対し同じ値が返るケースを考慮していない。そのような場合、探索を続けることができない。

Chapter 4

データ構造

データ構造 (**data structure**) とは、計算機のメモリ上にデータをどのように格納するかを示すものである。データ構造は長短があるため、問題に適したデータ構造を選択することが重要である。その際の重要な問いは、「選んだデータ構造は、どんな処理を効率よく行えるものか?」となる。

この章では C++ 標準ライブラリにおける最重要なデータ構造を紹介する。可能な限り、データ構造は実装するのではなく標準ライブラリのものを利用するのは、実装時間の短縮の上でも良い考えである。標準ライブラリに含まれないデータ構造については、後の章で触れる。

4.1 動的配列

動的配列 (**dynamic array**) とは実行中に長さを変更できる配列である。C++ で最も利用される動的配列は `vector` 型である。この型は通常の配列とほとんど同じように利用できる。

以下のコードは空の配列を作り、3つの要素を追加する:

```
vector<int> v;  
v.push_back(3); // [3]  
v.push_back(2); // [3,2]  
v.push_back(5); // [3,2,5]
```

その後、各要素は通常の配列同様に参照できる:

```
cout << v[0] << "\n"; // 3  
cout << v[1] << "\n"; // 2  
cout << v[2] << "\n"; // 5
```

`size` 関数は動的配列中の要素数を返す。以下のコードは動的配列の全要素を辿り内容を出力する:

```
for (int i = 0; i < v.size(); i++) {  
    cout << v[i] << "\n";  
}
```

動的配列の全要素を辿る処理は以下のように短く書くこともできる:

```
for (auto x : v) {
    cout << x << "\n";
}
```

back 関数は動的配列の最後の要素を返し、pop_back 関数は最後の要素を削除する:

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

以下のコードは 5 要素の動的配列を構築する:

```
vector<int> v = {2,4,2,5,1};
```

動的配列の別の構築法として、要素数と初期値を与える方法がある:

```
// 要素数10, 全要素0
vector<int> v(10);
```

```
// 要素数10, 全要素5
vector<int> v(10, 5);
```

動的配列の内部実装は通常の配列を使っている。動的配列の要素数が増えて事前確保した配列から溢れそうな場合、新たな配列を確保して要素をそちらにコピーする。しかしそのような事態は頻繁に起きるものではなく、push_back の平均時間計算量は $O(1)$ と考えてよい。

文字列型 (string) もほぼ vector 同様に使える動的配列である。加えて、いくつか文字列向けに特化した機能を持つ。文字列は + 演算子で連結可能である。substr(k, x) 関数は始点を k 文字目とし、長さを x とする部分文字列を持つ。find(t) 関数は、文字列中に部分文字列 t を含むなら、最初の登場位置を返す。

以下のコードはこれらの利用例である:

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

4.2 集合型

集合型 (**set**) は要素の集まりを保持するデータ構造である。集合に対する基本的な演算は、要素の追加・検索・削除である。

C++ の標準ライブラリには、集合型のための 2 つの実装がある。set 型は二分探索木を用いており、これらの演算が時間計算量 $O(\log n)$ で行える。unordered_set 型はハッシュを使っており、これらの演算を平均で時間計算量 $O(1)$ で行える。

どちらの実装を選ぶかは好みの問題に過ぎない場合もある。set 型の利点は、要素の並び順を保持しており、unordered_set にはない関数を持っていることである。一方 unordered_set は効率が良い。

以下のコードは集合型に整数を追加し、種々の演算を行う。insert 関数は集合型に要素を追加し、count 関数は集合型中で引数の値の出現回数を返し、erase 関数は集合型から指定された要素を削除する：

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

集合型は概ね vector と同じように扱えるが、カッコ [] を用いた添え字によるアクセスはできない。このコードは集合を作り、その要素数および各要素を順に表示するものである：

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

集合型の重要な特性として、すべての要素は互いに異なって (*distinct*) いなければならない。そのため、count 関数は 0 (集合中に要素がない) か 1 (集合中に要素がある) のいずれかしか返さない。また、insert 関数はすでに集合中に同じ値の要素がある場合、重複して同じ要素を追加することはない。以下のコードはこの特性を示す：

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

C++ の標準ライブラリには、multiset および unordered_multiset というデータ型がある。これらは set や unordered_set と似ているが、同じ値を持つ複数の要素を保持できる。例えば以下のコードは multiset 中に数値 5 を 3 つ格納している:

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

erase 関数は multiset 中で指定された値をもつ全要素を同時に取り除く:

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

もし要素を 1 つだけ削除したい場合、このようにすればよい:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

4.3 連想配列型

連想配列 (**map**) はキーと値の対の配列を管理するデータ構造である。通常の配列では、要素数を n とするとキーは連続した整数 $0, 1, \dots, n-1$ となるが、連想配列のキーは任意のデータ型で良く、連続した値である必要もない。

C++ の標準ライブラリは、set 同様に 2 種類の連想配列の実装を持つ。map 型は二分探索木に基づく実装で、要素の参照の時間計算量は $O(\log n)$ であり、unordered_map 型はハッシュに基づく実装で要素の参照の平均時間計算量は $O(1)$ である。

以下のコードはキーが文字列で値が整数であるような map を作成する:

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

キーに対応する値を参照しようとしたとき、連想配列にそのキーが格納されていないと、そのキーはデフォルト値を伴い自動的に生成される。例えば以下のコードはキー "aybabbu" に対し値 0 を追加する:

```
map<string,int> m;
cout << m["aybabbu"] << "\n"; // 0
```

count 関数でキーの存在は事前に確認できる:

```
if (m.count("aybaltu")) {
    // キーが存在する
}
```

以下のコードはすべてのキーと値の対を出力する:

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

4.4 イテレータと区間

C++ の標準ライブラリの多くの関数はイテレータを対象とする。イテレータは、データ型に対しある要素を指し示す変数である。

よく使われるイテレータは、データ型の格納する全要素の範囲を規定する `begin` と `end` である。`begin` はデータ型の最初の要素を指すイテレータで、`end` は最後の要素の後ろを示すイテレータである。これらは以下ようになる:

```

{ 3, 4, 6, 8, 12, 13, 14, 17 }
  ↑                               ↑
  s.begin()                       s.end()

```

イテレータの非対称性に注意すること。`s.begin()` が指すのはデータ構造内の要素であるが、`s.end()` はデータ構造の外側を指している。つまり、イテレータは半开区間を規定している。

区間の扱い

C++ の標準ライブラリの関数では、データ構造における要素の区間を表すのにイテレータを用いる。特に、データ構造内の全要素を処理したいとき、`begin` と `end` が関数に渡される。

例えば、以下のコードは `vector` を `sort` 関数でソートし、`reverse` 関数で並びを反転させ、最後に `random_shuffle` 関数で並びをシャッフルさせる。

```
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
random_shuffle(v.begin(), v.end());
```

これらの関数は通常の配列にも適用可能である。先の例では、イテレータの代わりにポインタを渡すことで配列に同じ処理を行える:

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

集合型のイテレータ

イテレータは集合の要素のアクセスにも用いる。以下のコードは、set 中で最小の要素を指すイテレータ `it` を作成する:

```
set<int>::iterator it = s.begin();
```

このコードは短く以下のようにも書ける:

```
auto it = s.begin();
```

イテレータが指す要素には、`*` 記号を使いアクセスできる。例えば、以下のコードは set 中で最小の要素を出力する。

```
auto it = s.begin();
cout << *it << "\n";
```

イテレータは、演算子 `++` で次の要素、`--` で手前の要素を指すように移動させることができる。

以下のコードは全要素を昇順で出力する:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

以下のコードは set 中で最大の要素を出力する:

```
auto it = s.end(); it--;
cout << *it << "\n";
```

`find(x)` 関数は、値 x を持つ要素を指すイテレータを返す。もし集合中に要素 x が見つからない場合、返すイテレータは `end` と一致する:

```
auto it = s.find(x);
if (it == s.end()) {
    // xは見つからなかった
}
```

`lower_bound(x)` 関数は、集合中で値が少なくとも x である最小の要素を指すイテレータを返し、`upper_bound(x)` 関数は、値が x より大きい最小の要素を指すイテレータを返す。いずれも、そのような要素が存在しない場合は `end` と一致するイテレータを返す。これらの関数は `unordered_set` では利用できない。`unordered_set` は要素の並び順を保持しないためである。

例えば、以下のコードは x に最も近い要素を探す:

```

auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\n";
} else {
    int a = *it; it--;
    int b = *it;
    if (x-b < a-x) cout << b << "\n";
    else cout << a << "\n";
}

```

このコードは集合が空でないことを前提とする。このコードはイテレータ `it` を使い要素を探索している。まず、このイテレータは値が少なくとも x である最小の要素を指す。もし `it` が `begin` と一致するなら、この要素が x と最も近い。もし `it` は `end` と一致するなら、集合の最大値の要素が x と最も近い。これらいずれでもない場合、 x に一番近い要素は、`it` かその一つ手前の要素である。

4.5 その他のデータ構造

Bitset

bitset は 0/1 の 2 値の配列である。例えば、以下のコードは 10 要素の **bitset** を作成する:

```

bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0

```

bitset を使う利点は、通常の配列よりメモリ消費量が少ないことである。というのも、**bitset** の各要素はメモリを 1bit しか消費しない。

例えば n 個のビットを `int` 型の配列に個別に格納しようとする、 $32n$ ビットのメモリを使用するが、**bitset** の場合は n ビットしか消費しない。それに加えて、**bitset** はビット演算と同様の演算子を用いて、ビットの集合向けに高速なビット演算を行える。

以下のコードは上記の例と同様の **bitset** を作る別の方法を示す:

```

bitset<10> s(string("0010011010")); // from right to left
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0

```

count 関数は bitset 中で 1 の値の要素数を示す:

```
bitset<10> s(string("0010011010"));  
cout << s.count() << "\n"; // 4
```

以下のコードはビット演算の例を示す:

```
bitset<10> a(string("0010110110"));  
bitset<10> b(string("1011011000"));  
cout << (a&b) << "\n"; // 0010010000  
cout << (a|b) << "\n"; // 1011111110  
cout << (a^b) << "\n"; // 1001101110
```

双方向キュー

双方向キュー (**deque**) は両端におけるサイズを効率よく変えられる動的配列である。vector 同様、deque は push_back と pop_back 関数を持つ。それに加えて、vector が持たない push_front と pop_front 関数も利用可能である。

deque は以下のように利用する:

```
deque<int> d;  
d.push_back(5); // [5]  
d.push_back(2); // [5,2]  
d.push_front(3); // [3,5,2]  
d.pop_back(); // [3,5]  
d.pop_front(); // [5]
```

deque の内部実装は vector よりは複雑なため、処理は vector よりは遅い。ただし、両端における要素の追加削除の時間計算量は $O(1)$ である。

スタック

スタック (**stack**) は、要素を最上段に追加することと最上段の要素を取り除くことの 2 つの処理を時間計算量 $O(1)$ で行えるデータ構造である。スタックにおいてアクセスできる要素は最上段のみである。

以下のコードはスタックの使用例である。

```
stack<int> s;  
s.push(3);  
s.push(2);  
s.push(5);  
cout << s.top(); // 5  
s.pop();  
cout << s.top(); // 2
```


キュー

キュー (**queue**) は先頭の要素を取り除くことと末尾に要素を追加することの 2 つの処理を時間計算量 $O(1)$ で実行する機能を持つデータ構造である。キューにおいては先頭と末尾の要素しかアクセスできない。

以下のコードはキューの使用例である:

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

優先度付キュー

優先度付キュー (**priority queue**) は要素の集合を保持するデータ構造である。備えている機能は、要素の追加と、最大値もしくは最小値の要素の参照・削除である。要素の追加・削除は時間計算量 $O(\log n)$ 、参照は $O(1)$ で行える。

順序付 set も優先度付キューが備えるのと同等の処理を行えるが、優先度付キューの方が軽い実装になっており定数倍実行時間が短い。順序付 set は二分探索木で実装されているが、優先度付キューはより単純なヒープで実装されていることがこの違いの要因である。

C++ の標準ライブラリの優先度付キューは、標準では要素を降順に並べる。つまり要素の最大値の参照・削除ができる。以下のコードは実例を示す:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

もし最大値ではなく最小値の参照・削除ができる優先度付キューが必要な場合、このように実現できる:

```
priority_queue<int, vector<int>, greater<int>>> q;
```

ポリシーベースのデータ構造

g++ コンパイラは、C++ の標準ライブラリに含まれないデータ構造もサポートしている。それらはポリシーベースのデータ構造と呼ばれる。これらのデータ構造を利用するには、以下のコードを追加する:

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

その後、`indexed_set` 型が利用可能になる。この型は `set` と似ているが、配列のように添え字によるアクセスも利用できる。`int` 型の要素を格納するデータ型の定義は、以下のようになる:

```
typedef tree<int,null_type,less<int>,rb_tree_tag,
            tree_order_statistics_node_update> indexed_set;
```

その後、以下のように集合を作ることができる:

```
indexed_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

この集合型の特徴は、ソート済み配列とみなし添え字による参照が可能なことである。`find_by_order` 関数は指定された位置へのイテレータを返す:

```
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7
```

`order_of_key` 関数は逆に値に対応する位置（配列の添え字）を返す:

```
cout << s.order_of_key(7) << "\n"; // 2
```

集合中に指定した値の要素が存在しない場合、その値が存在していたら入るであろう位置を返す:

```
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

両関数は対数時間で動作する。

4.6 ソートとの比較

問題を解く際、しばしば特殊なデータ構造と単なるソートいずれでも解ける場合がある。両者は、時間計算量には現れない効率の顕著の違いを生じることがある。

例として、共に n 要素を保持する 2 つのリスト A と B があるとする。ここ

で両リスト共に含まれる要素の数を求める問題を考えよう。例えばリストが

$$A = [5, 2, 8, 9, 4] \quad \text{かつ} \quad B = [3, 2, 9, 5],$$

であれば、両リストに 2, 5, 9 が含まれるので答えは 3 である。

単純な解法としては、両リストのペアを総当たりする時間計算量 $O(n^2)$ の解法がある。以下、より効率的なアルゴリズムを考えよう。

アルゴリズム 1

A に現れる要素からなる集合を作り、次に B の各要素について先ほどの集合に含まれるかを確認する。この方法は A の要素はすでに集合になっているため、要素を効率よく検索でき高速に動作する。set 型を用いると、このアルゴリズムの時間計算量は $O(n \log n)$ となる。

アルゴリズム 2

上記アルゴリズムにおいて、要素の順序を保存しておく必要はない。よって set 型を unordered_set 型に差し替えれば、それだけで効率が改善し、時間計算量は $O(n)$ となる。

アルゴリズム 3

別のデータ構造を持ち込むのではなくソートだけで解いてみることにする。まず A, B 両方の配列をソートしておく。次に、両配列を先頭から同時に探索し、共通する要素を探し出していく。このアルゴリズムでは、ソートの時間計算量は $O(n \log n)$ であり、その後の共通要素の検出は $O(n)$ である。よって全体の時間計算量は $O(n \log n)$ となる。

効率の比較

以下の表は、要素が $1 \dots 10^9$ の範囲のランダムな整数である長さ n の配列に対し、各アルゴリズムがどの程度効率よく動くかを示す：

n	アルゴリズム 1	アルゴリズム 2	アルゴリズム 3
10^6	1.5 s	0.3 s	0.2 s
$2 \cdot 10^6$	3.7 s	0.8 s	0.3 s
$3 \cdot 10^6$	5.7 s	1.3 s	0.5 s
$4 \cdot 10^6$	7.7 s	1.7 s	0.7 s
$5 \cdot 10^6$	10.0 s	2.3 s	0.9 s

アルゴリズム 1 と 2 は用いたデータ型が異なる以外同じ処理をしている。この問題においてはこの違いは重要で、アルゴリズム 2 はアルゴリズム 1 の 4 ~ 5 倍高速である。

しかし、最も効率が良いのはソートを用いたアルゴリズム 3 であり、アルゴリズム 2 の半分以下の時間しかかかっていない。アルゴリズム 1 と 3 の時間計算量は共に $O(n \log n)$ であるが、後者の方が 10 倍は速い。これはソート

が非常に単純な手続きであり、かつソート後の処理は線形時間で済むためである。一方アルゴリズム 1 は常時複雑な二分探索木の維持をしなければならないため、この時間差が生じた。

Chapter 5

全探索

全探索 (**Complete search**) はほとんどのアルゴリズムの問題を解くことができる手段である。そのアイデアは、あり得る可能性を総当たりですべて列挙し、その中で問題に応じて最適値を求めたり組み合わせの総数を求めたりするというものである。

全探索は一般的に実装が容易であり、かつ正しい解にたどり着きやすいため、十分な時間があるならば良いテクニックである。全探索では遅すぎる場合は、別のテクニック、例えば貪欲法なり動的計画法なりが必要となる。

5.1 部分集合の生成

まず n 要素の集合から、あり得る全部分集合を列挙する問題を考えよう。例えば $\{0, 1, 2\}$ の部分集合は $\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}$ である。部分集合の生成には 2 つの定番手法として再帰的な探索と、ビット演算の活用がある。

方法 1

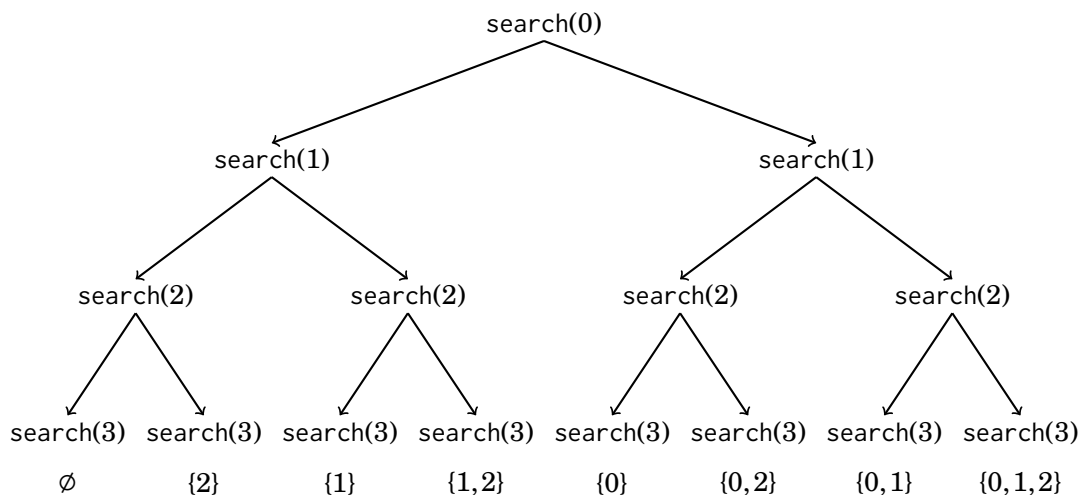
全部分集合を探索するエレガントな方法として再帰を用いる方法がある。以下の `search` 関数は集合 $\{0, 1, \dots, n-1\}$ の部分集合を生成する。

この関数は、部分集合の要素を含む配列 `subset` を管理していく。探索はこの関数を引数 0 で呼び出して始める：

```
void search(int k) {
    if (k == n) {
        // 部分集合に関して処理を行う
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

関数 `search` が引数 k で呼び出されると、その際に部分集合の要素として k を含む場合と含まない場合を考慮し、それぞれに対して引数を $k+1$ として再帰的に処理を行う。 $k=n$ となると n 未満の全要素について処理が終わったことがわかるので、その時点で部分集合が生成されたと判断できる。

以下の樹形図は、 $n=3$ の場合の関数呼び出しを表したものである。この関数は常に左側の枝 (k が部分集合に含まれない) を探索したのち右側の枝 (k が部分集合に含まれる) を探索する。



方法 2

部分集合群を生成する別の方法は、整数におけるビット表現を活用することである。 n 要素の集合の部分集合を n ビットの並びとして表現する。すなわち $0 \dots 2^n - 1$ の範囲の整数に対応付ける。ビット列中 1 がセットされた箇所は、対応する要素が部分集合中に含まれていることを意味する。

良くある対応付けは、末尾のビットを要素 0、末尾から 2 番目のビットを要素 1、というように対応付けるものである。例えば整数値 25 のビット表現は 11001 となるが、これは部分集合 {0,3,4} と対応付けられる。

n 要素の部分集合群は、以下のコードで探索できる:

```

for (int b = 0; b < (1<<n); b++) {
    // 部分集合を処理する
}

```

以下のコードは、ビット列に対応する部分集合の生成法を示している。各部分集合を示すビット列に対応して、実際の要素を含む配列を生成している。

```

for (int b = 0; b < (1<<n); b++) {
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)) subset.push_back(i);
    }
}

```

5.2 順列の生成

続けて、 n 要素のすべての順列を生成する問題を考える。例えば、 $\{0,1,2\}$ の順列は $(0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,0,1), (2,1,0)$ である。こちらも 2 つの定番アプローチとして、再帰を用いる方法と順次順列を生成していく方法がある。

方法 1

部分集合同様、順列も再帰を使って生成できる。以下の関数 `search` は集合 $\{0,1,\dots,n-1\}$ の順列を生成し、配列 `permutation` に格納していく。この再帰関数は、配列が空の状態が始まる：

```
void search() {
    if (permutation.size() == n) {
        // 順列に関し処理を行う
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

関数が呼び出されるたびに `permutation` に要素が追加されていく。配列 `chosen` は、どの要素がすでに順列に含まれているかを示す。`permutation` の大きさが元の集合と一致すると処理は完了し、順列が生成された状態となる。

方法 2

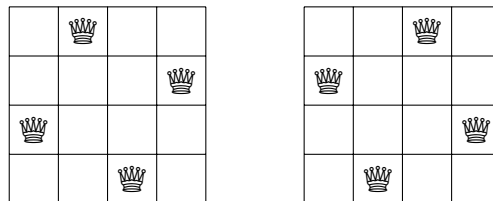
順列を生成する別の方法は、まず $\{0,1,\dots,n-1\}$ から初めて、辞書順で次の順列を順次生成していく方法である。C++ 標準ライブラリにある `next_permutation` 関数がこの手続きを提供している：

```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // 処理を行う
} while (next_permutation(permutation.begin(), permutation.end()));
```

5.3 バックトラック法

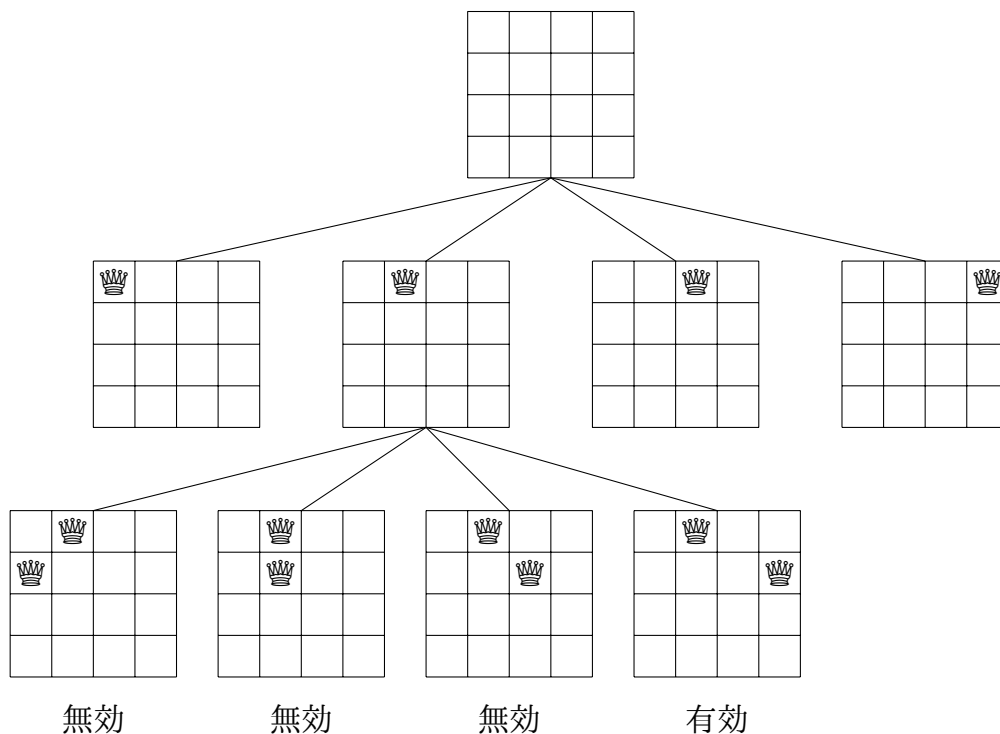
バックトラック法 (**backtracking**) とは、初期状態から 1 手ずつ再帰的に先に進めていき、最終的に異なる状態をすべてたどる方法である。

例として、 $n \times n$ の大きさのチェス盤に n 個のクイーンの駒を互いに攻撃できない位置に置く置き方を数え上げる問題を考える。例えば $n = 4$ の場合次の 2 つの解がある:



この問題は行ごとに駒を配置していくバックトラック法で解くことができる。より正確に書くと、各列にすでに置いた駒と互いに攻撃しあわないような位置に新たに 1 箇所駒を追加していく。 n 個の駒をすべて配置したらそれが解となる。

例えば $n = 4$ の場合、バックトラック法で生成される解の一部は下記の通りとなる:



図における最下段の 4 通りの例において、先頭の 3 つの構成はクイーンが互いに互いに攻撃できるため無効である。しかし 4 つ目の構成は有効であり、残り 2 つのクイーンを置いていくと完全な解まで探索を拡張できる。残り 2 つのクイーンの置き方は一通りしかない。

このアルゴリズムはこのように実装できる:


```

void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}

```

この探索は `search(0)` を呼ぶことで開始する。盤面の大きさを n とすると、このコードは求める解を計算し `count` に格納する。

このコードでは、列および行が 0 から $n-1$ まで番号が振られているものとして計算を行う。

関数 `search` が引数 y を伴って呼び出されると、この関数は y 行目にクイーンを 1 つ置き、自身の関数を引数 $y+1$ として再帰的に呼び出す。そのため、 $y=n$ に到達すると、解が 1 つ見つかったとみなすことができ、変数 `count` がインクリメントされる。

配列 `column` はクイーンが配置された列の情報を保存するものであり、配列 `diag1` と `diag2` はクイーンが配置された斜め方向の列の情報を保存するものである。すでにクイーンの配置された列や対角上にクイーンを追加することはできない。例えば、 4×4 の盤面上を以下のように番号を割り当てる：

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

列

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

対角 1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

対角 2

$q(n)$ を $n \times n$ の盤面に n 個のクイーンを置く置き方の総数とする。先ほどのバックトラッキングにより、例えば $q(8)=92$ であることがわかる。 n が増えると、探索は非常に遅くなる。というのも、総数が指数関数的に大きくなるためである。例えば、 $q(16)=14772512$ をこの方法で計算すると最近の計算機でも 1 分程度かかる。¹

5.4 探索の枝刈り

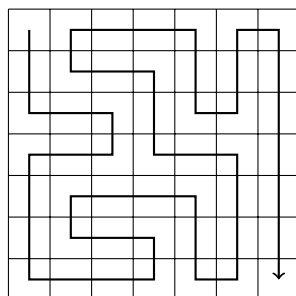
しばしば探索木を枝刈りすることで、バックトラッキングの最適化が可能である。そのアイデアは、アルゴリズムに判断を加え、今の探索を継続しても

¹ 大きな $q(n)$ を効率的に計算する手法は知られていない。現在の記録は、2016 年に計算された $q(27)=234907967154122528$ である [55]。

求める解にたどり着きそうにない場合、できるだけ早く気づき探索を打ち切ることである。このような最適化は探索の効率に著しく影響する場合がある。

ここで、 $n \times n$ のグリッドにおいて、左上角のセルから右下角のセルに全セルを1回ずつ通って到達するパスの組み合わせを求める問題を考えてみよう。

例えば、 7×7 のグリッドにおいてそのようなパスは 111712 通りある。下記はその一例である：



以下アルゴリズムを考える上で適度な難易度なので、 7×7 のケースに着目していく。まず単純なバックトラッキングアルゴリズムから始め、そこから探索を枝刈りできるか考察することで徐々に最適化していこう。

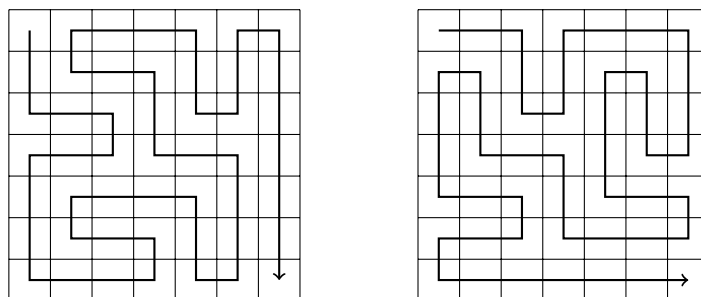
基本的なアルゴリズム

最初のバージョンは、何も最適化せず、単純に可能なパスを全列挙してその数を数えるものとした。

- 実行時間: 483 秒
- 再帰呼び出しの回数: 760 億回

最適化 1

どんな解も、最初のステップは下か右である。ある解に対して、対角に対して線対称な2つのパスが存在する。例えば、以下の2つのパスは対称である。

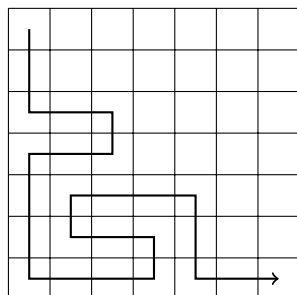


そのため、最初の移動は下（または右）と決め打ちし、最後に得られた値を倍にしよう

- 実行時間: 244 秒
- 再帰呼び出しの回数: 380 億回

最適化 2

もしパスが全セルを辿る前に右下角に到達してしまった場合、以後の処理を続けても求めるパスにならないことは明らかである。以下のパスはその例である:

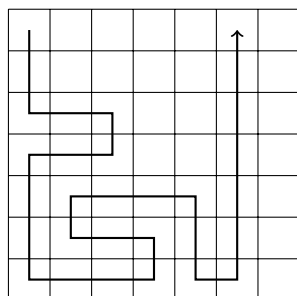


この考察より、右下角に早く到達した場合、探索を打ち切ることができる。

- 実行時間: 119 秒
- 再帰呼び出しの回数: 200 億回

最適化 3

パスが途中で周辺部に到達して、次に左か右に曲がるとき、グリッドは未到達セルを含む 2 つのパートに分割される。例えば、以下の状況では、パスは左か右に曲がることができる。

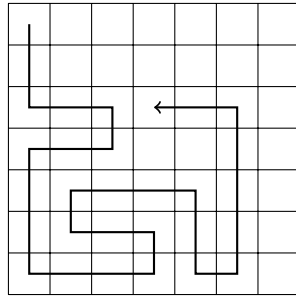


この例では、もはや全セルを辿ることはできないため、探索を打ち切ることができる。この最適化は非常に効果的である:

- 実行時間: 1.8 秒
- 再帰呼び出しの回数: 2.21 億回

最適化 4

最適化 3 のアイデアをさらに一般化する。パスが前に伸ばすことができず、左右どちらかに曲げることが可能な場合、グリッドは 2 つの未到達パートに分割される。例えば以下のようなパスを考える:



この例でも、全セルを辿ることができないことは明らかなので、探索を打ち切ることができる。この最適化により、探索は非常に効率的になった：

- 実行時間: 0.6 秒
- 再帰呼び出しの回数: 6900 万回

この辺で最適化を止め、効果を振り返ってみよう。元のアлゴリズムの実行時間は 483 秒だが、度重なる最適化により、たった 0.6 秒まで短縮した。すなわち 1000 倍近く高速化したことになる。

このようなことはバックトラッキングではよくあることである。というのも探索木は通常非常に大きなものであり、ちょっとした考察でも効率的に枝刈りできることがあるためである。特にアルゴリズムの最初のステップにおける最適化は非常に効率的である。例えば探索木のトップにおける最適化が相当する。

5.5 半分全列挙

半分全列挙²は探索空間を同サイズの 2 つの空間に区切るテクニックである。分けた個々の空間に対して探索を行い、最後の両者の結果を組み合わせる。

このテクニックが有効なのは、探索結果を組み合わせることが効率的にできる手段がある場合である。そのような場合、小さな 2 つの空間の探索を大きな 1 つの空間の探索より短い時間で実行できる。特に、 2^n 個の要素を半分全列挙により $2^{n/2}$ 個に分けることができる。

例として、 n 個の数値からなるリストと数値 x が与えられたとき、和が x となる部分集合の存在を判定する問題を考える。例えば、リストが $[2, 4, 5, 9]$ で $x = 15$ の場合、 $2 + 4 + 9 = 15$ より解として $[2, 4, 9]$ を選ぶことができる。しかし、同じリストに対して $x = 10$ となる和を構成することはできない。

単純なアルゴリズムは、リストの要素の全部分集合を探索し、各部分集合の和が x になるか判定することである。このアルゴリズムの実行時間は、部分集合が 2^n 通りあることから $O(2^n)$ となる。しかしながら、半分全列挙を用いると、より効率的に $O(2^{n/2})$ 時間のアルゴリズムを構成できる³。なお、 $O(2^n)$ と $O(2^{n/2})$ は異なる計算量であることに注意すること。なぜなら $2^{n/2}$ は $\sqrt{2^n}$ と等しいからである。

²訳注: Meet in the middle はセキュリティ分野では中間一致攻撃と訳されることがあるが、ここでは競技プログラミングで使われる単語としてあえてこちらを用いた

³このアイデアは 1974 年に E. Horowitz と S. Sahni によって紹介された [39]。

このアルゴリズムのアイデアは、元のリストを半分の要素数を持つ2つのリスト A と B に分けることである。まず最初の探索では、 A の全部分集合を列挙し、その和をリスト S_A に格納する。同様に、2回目の探索ではリスト B から部分集合の和のリスト S_B を構成する。その後、 S_A の要素と S_B の要素を選び、その和が x となる場合があるかどうか判定する。そのような要素があるということは、元のリストにおいても和が x となる要素の選び方が可能であると言える。

例えば、リストを $[2, 4, 5, 9]$ 、 $x = 15$ とする。まずリストを $A = [2, 4]$ と $B = [5, 9]$ の2つに分割する。その後、それぞれに対応して和のリスト $S_A = [0, 2, 4, 6]$ と $S_B = [0, 5, 9, 14]$ を作る。このケースでは、和が $x = 15$ となるものは構成可能である。というのも、 S_A は和 6 を含み、 S_B は和 9 を含むため、 $6 + 9 = 15$ を構成できるためである。対応するリストの要素は $[2, 4, 9]$ である。

このアルゴリズムの時間計算量は $O(2^{n/2})$ である。というのも、両リスト A と B は約 $n/2$ 要素を持っており、その部分集合の総和を求めてリスト S_A と S_B を構成するのにそれぞれ $O(2^{n/2})$ にかかるためである。その後、 S_A と S_B から和が x となるものを構成できるか判定するのは $O(2^{n/2})$ でできる。

Chapter 6

貪欲アルゴリズム

貪欲アルゴリズムは、そのつど最良に見える手を選択していく形で解法を構成する手法である。

貪欲アルゴリズムは選択結果を巻き戻すことはせず、直接最終的な解を構成する。そのため、貪欲アルゴリズムは通常非常に効率的である。

貪欲アルゴリズムの設計の難しさは、問題に対し必ず最適化を生成できるような貪欲な戦略を探すところにある。局所的に最適な選択が、全体でも最適でなければならない。そのため、貪欲解法で正答できると判断するのはしばしば難しい場合がある。

6.1 コインの問題

最初の例として、コインの集合が与えられたとき、和が n となるようなコインの部分集合を求める問題を考えよう。それぞれのコインの価値は $\text{coins} = \{c_1, c_2, \dots, c_k\}$ であるとし、それぞれのコインは好きなだけ利用できるとする。部分集合を構成する際最小の枚数はいくつになるだろうか？

例えば、ユーロにおけるコインの金額の種類は

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

であり、和 x が 520 となるケースでは、コインは最小 4 枚必要である。最適な解はコインを $200 + 200 + 100 + 20 = 520$ となるように選択することである。

貪欲アルゴリズム

この問題に対するシンプルな貪欲アルゴリズムは、和が x に達するまで、できるだけ大きな金額のコインを使っていくことである。このアルゴリズムは上記の例ではうまく動作する。まず 200 セントのコインを 2 枚選択し、100 セントのコインを 1 枚選択して最後に 20 セントのコインを選択すればよい。しかしこのアルゴリズムは常に有効なのだろうか？

コインの種類がユーロと同じであれば、この貪欲アルゴリズムは常に正解を返す。つまり最小のコイン枚数の解を得ることができる。このアルゴリズムの正しさは下記の通り示すことができる：

まず、1, 5, 10, 50, 100 のコインは最適解において高々 1 枚しか選択しえない。というのも、これらのコインが 2 枚あるのであれば、より金額の高い 1

枚と置き換えることでより良い解を得られるからである。例えば、もし解が $5+5$ を含むのであれば、 10 のコインに置き換えることができる。

同様に、 $2,20$ のコインも最適解では高々 2 枚までしか含まれない。というのも、 $2+2+2$ の構成は $5+1$ に、 $20+20+20$ の構成は $50+10$ に置き換えられるからである。さらに、最適解は $2+2+1$ や $20+20+10$ という構成も持ちえない。これらは 5 や 50 のコイン 1 枚に置き換えられるためである。

これらの考察より、各コイン x に対して、 x を用いずに和が x 以上となる組み合わせの最適化を構成できないことがわかる。例えば、 $x=100$ を例にすると、これより小さい価値のコインで表せる最適解が最大枚数となるケースは $50+20+20+5+2+2=99$ となる。そのため、常に最大価値のコインを選択する貪欲アルゴリズムは最適解を生成できる。

この例は、アルゴリズム自体は単純でもそれが最適であることを示すのが比較的難しいことを示す好例である。

一般的な例

コインの組み合わせを任意に設定できる場合において、この貪欲アルゴリズムは最適解を生成できるとは限らない。

これは前述の貪欲アルゴリズムが非最適解を返す反例を持って示すことができる。コインの種類が $\{1,3,4\}$ で和を 6 としたい場合、最適解は $3+3$ だが、貪欲アルゴリズムでは $4+1+1$ となってしまう。

この一般的な例が貪欲解法で解けるかどうかは分かっていない¹。しかし、7章で例を示す通り、動的計画法でこの問題は効率よく解くことができる。

6.2 スケジューリング

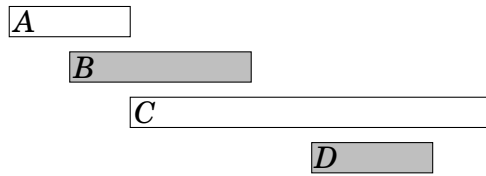
多くのスケジューリング問題は貪欲アルゴリズムで解くことができる。ここで古典的問題を考えよう。 n 個のイベントがあり、開始時間と終了時間が与えられる。最適にイベントをスケジューリングしたとき、最大でいくつのイベントを含むことができるか。ただしイベントの開催時間の一部だけ参加するということはできない。

例えば以下のイベント例を考える：

イベント	開始時間	終了時間
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

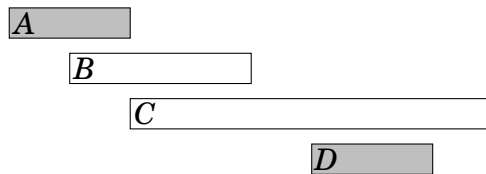
この例ではイベントの最大数は 2 である。例えば下記のようにイベント *B* と *D* を選ぶことができる：

¹しかし、前述の貪欲解法で解けるコイン種別かどうかを多項式時間で判定することはできる [53]。

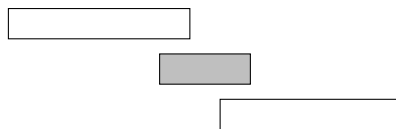


アルゴリズム 1

最初のアイデアは、短いイベントをできるだけ選んでいく方法である。サンプルケースでは、このアルゴリズムは以下のようにイベントを選ぶ:



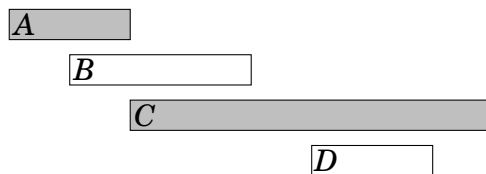
しかし、短いイベントを選ぶ方法は常に正解とは限らない。例えば次の例では正解を得られない:



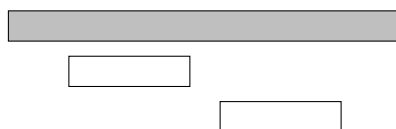
この例では、短いイベントを選ぶと1つしかイベントを選択できない。しかし明らかに長いイベントを2つ選ぶ方が正解である。

アルゴリズム 2

次のアイデアは、開始時間ができるだけ早いものをできる限り選んでいく方法である。このアルゴリズムでは以下のようにイベントが選択される:



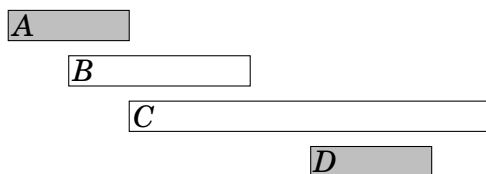
しかし、このアルゴリズムに対しても反例が存在する。以下の例では、このアルゴリズムでは1つしかイベントを選択できない:



もし最初のイベントを選ぶと、それ以上他のイベントを選べない。しかし、最初のイベントの選択をあきらめれば、残り2つのイベントを選ぶことができる。

アルゴリズム 3

3つ目のアルゴリズムは、終了時間ができるだけ早いものをできる限り選んでいく方法である。このアルゴリズムでは以下のようにイベントを選択する:



このアルゴリズムは常に最適解を返すことがわかる。その理由は、終了時間の早いイベントを最初を選ぶことが常に最適な選択であるからである。そのイベントの後は、同様の手順で次のイベントを最適に選択していくことを、それ以上イベントが選べなくなるまで繰り返す。

このアルゴリズムがうまく動くことを示す1つの方法は、終了時間が早いイベントを選んだ時何が起きるかを考えることである。その場合、以後どれだけ最適な選び方をしても、最初のイベントで終了時間が早いものを選んだ場合とせいぜい同数にしかない。よって、終了時間の遅いものを選ぶことがより良い解となることはなく、この貪欲アルゴリズムは正しい。

6.3 タスクと締切

次に、別の問題を考えよう。 n 個のタスクについて、実行時間と締切時刻が与えられる。これらのタスクを処理する順番を考えよう。各タスクについて、締切時刻を d 、タスクを終了した時刻を x とすると、 $d-x$ ポイントの点を得られるものとする。総得点の最大値はどうなるだろうか？

例えば、タスクが以下の通りだとする:

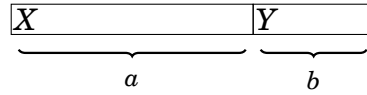
タスク名	実行時間	締切時刻
A	4	2
B	3	5
C	2	7
D	4	5

この例では、最適なスケジュールは以下ようになる:

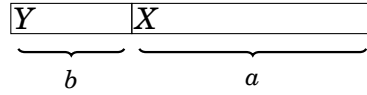


この回答では、Cで5ポイント、Bで0ポイント、Aで-7ポイント、Dで-10ポイント得ることができ、総得点は-10ポイントとなる。

驚くことに、この問題の最適解は締切時刻には全く関係ない。正しく動作する貪欲な戦略は、単にタスクを実行時間で昇順にソートすることである。その理由は、2つの連続するタスクにおいて前者が後者より実行時間が長い場合、それらを交換することでより良い解を得られるためである。例えば、以下のスケジュールを考えよう:



ここで $a > b$ なので、両者は交換するべきである:



ここでタスク X により得られるポイントは b 減少し、タスク Y により得られるポイントは a 増加する。よって総得点は $a - b > 0$ だけ増加する。最適解では、2つの連続するタスクでは常に先行するタスクには実行時間が短いほうが割り当てられる。よって、タスクは実行時間で昇順ソートした順で処理されなければならない。

6.4 和の最小化

次の問題として、 n 個の数値 a_1, a_2, \dots, a_n が与えられたとき、和

$$|a_1 - x|^c + |a_2 - x|^c + \dots + |a_n - x|^c.$$

を最小化するような数 x を求める問題を考えてみよう。ここでは $c = 1$ と $c = 2$ の場合のみ考える。

$c = 1$ の場合

この場合、以下の和を最小化する問題となる。

$$|a_1 - x| + |a_2 - x| + \dots + |a_n - x|.$$

例えば、入力の数値が $[1, 2, 9, 2, 6]$ であれば、最適解は $x = 2$ であり、その時の和は下記のとおりとなる。

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

一般に、 x の最適値は数値群の中央値（注：要素をソートした後の中央の値）となる。例えば、入力の数値群 $[1, 2, 9, 2, 6]$ はソート後 $[1, 2, 2, 6, 9]$ となるので、中央値は 2 である。

中央値が最適値となるのは、もし x が中央値より小さい場合、和は x を大きくする毎に小さくなり、逆に x が中央値より大きいならば、和は x をへらほど小さくなる。よって最適解は x を中央値とすることである。 n が偶数の場合中央値は 2 つあるが、その場合両方の値、および両者の間にあるあらゆる数が最適解となる。

c = 2 の場合

この場合、以下の和を最小化する問題となる。

$$(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2.$$

例えば、入力値が [1, 2, 9, 2, 6] の場合、 $x = 4$ が最適解であり、その時の和は下記のようになる。

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46.$$

一般に、 x の最適解は入力値の平均値となる。この例では平均値は $(1 + 2 + 9 + 2 + 6)/5 = 4$ である。この結果は、和を以下のように表現するとわかる：

$$nx^2 - 2x(a_1 + a_2 + \cdots + a_n) + (a_1^2 + a_2^2 + \cdots + a_n^2)$$

最後の項は x の値によらないので無視できる。残りの項は、 $s = a_1 + a_2 + \cdots + a_n$ とおくと $nx^2 - 2xs$ と表せる。これは上に開いた放物線で、 $x = 0$ と $x = 2s/n$ が根となる。そして最小値は $x = s/n$ のときに得られるが、これは先ほど計算した平均値そのものである。

6.5 データ圧縮

バイナリコードは文字列を構成する個々の文字に、ビット列からなる符号表を割り当てる。ここで、文字列の個々の文字を符号表を用いて対応するバイナリコードに置き換えることで、文字列を圧縮できる。例えば、文字 A-D に対し以下のようなバイナリコードの対応表があるとする：

文字	符号
A	00
B	01
C	10
D	11

各符号長は等しいため、これは固定長の符号ということになる。例えば、文字列 AABACDACA は符号表により圧縮すると

000001001011001000

となる。この符号表では、圧縮後の文字列は 18 ビットとなる。しかし、この文字列は各符号の長さにはばらつきを許容する 可変長の符号を用いることでより良く圧縮できる。頻繁に現れる文字に短い符号、稀にしか現れない文字に長い符号を割り当ててみる。すると、上記文字列に対応する最適な符号表はこのようになる：

文字	符号
A	0
B	110
C	10
D	111

最適な符号表では、文字列をできるだけ短く圧縮する。この例では、圧縮後の文字列は

001100101110100,

であり、固定長の符号表では 18 ビットだった文字列がこちらではたった 15 ビットで済んでいる。つまり、より良い符号表により圧縮後の文字列が 3 ビット節約できている。

この符号表において、ある符号が他の符号の **prefix** になっていてはならない。例えば、符号表に 10 と 1011 が混在してはならない。その理由は、圧縮後の文字列から元の文字列を復元可能でなければならないためである。もしある符号が他の符号の **prefix** だと、復元が正しく行えない場合が生じてしまう。例えば、以下の符号表は不正である：

文字	符号
A	10
B	11
C	1011
D	111

この符号表では、圧縮後の文字列 1011 から、元の文字列が AB と C のどちらであるか判断できない。

ハフマン符号

ハフマン符号² は文字列に対し最適な符号表を生成する貪欲アルゴリズムである。このアルゴリズムは、文字列中の文字の登場頻度に応じ二分木を作る。そして各文字の符号は根から各文字に対応する頂点までのパスに対応付けられる。木を辿る過程で、二分木の左の子を辿るとき 0、右の子を辿るとき 1 に対応付けられる。

初期状態では、文字列中の各文字は単独の頂点をなし、その重さは文字列中の登場回数とする。1 ステップごとに、重さが最少の頂点 2 つを選び、それら 2 つを子頂点とする親頂点を生成する。親頂点の重さは、子頂点の重さの和である。この処理を全頂点が連結するまで繰り返す。

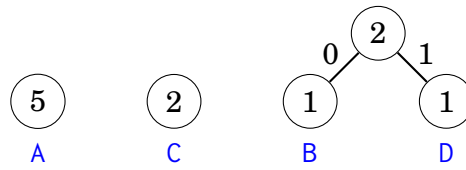
以下、ハフマン符号により文字列 AABACDACA の符号表を作る過程を見ていこう。初期状態では、文字列中の文字に対応し以下の通り 4 つの頂点がある：



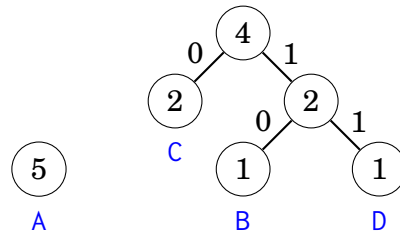
文字 A に対応する頂点の重さは 5 である。というのも、文字 A は文字列中で 5 回現れるためである。他の頂点の重さも同様に計算される。

最初のステップでは、文字 B と D に対応する頂点がともに重さが 1 で最小なので、それらを連結しよう。結果は下記のとおりである。

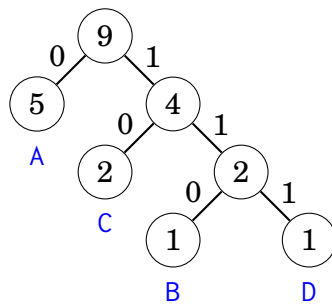
²D. A. Huffman はこの手法を大学の授業の時間割を考えている時に発明し、1952 にこのアルゴリズムを発表した [40]。



次に、重さ 2 の頂点同士を連結する:



最後に、残った 2 つの頂点を連結する:



これで頂点はすべて単一の木を構成し、対応する符号表も完成した。この木より、以下の符号表が得られる:

文字	符号
A	0
B	110
C	10
D	111

Chapter 7

動的計画法

動的計画法 (**Dynamic programming**) は、全探索の正確さと貪欲アルゴリズムの効率を組み合わせたテクニックである。動的計画法が適用できるのは、元の問題を個々に解くことができる部分問題に分割できる場合である。

動的計画法には2つの用途がある:

- 最適解の発見: できるだけ大きい解、もしくはできるだけ小さい解を見つきたい。
- 解の数え上げ: 条件を満たす解の総数を計算したい。

まず、動的計画法が最適解を探す例と、数え上げを行う例を見ていこう。

動的計画法の理解は、すべての競技プログラミング参加者のマイルストーンである。動的計画法の基本的アイデアは非常にシンプルだが、個々の問題にどのように適用するかという点にむずかしさがある。本章では、スタート地点として良い題材となる古典的問題を紹介していこう。

7.1 コイン問題

ここではすでに6章で見た問題を再度題材とする。価値が $\text{coins} = \{c_1, c_2, \dots, c_k\}$ であるコインが与えられ、目的の金額 n が与えられたとき、総額が n となる最小枚数のコインを求める問題である。

6章では、我々はこの問題を常に選ぶうる最大金額のコインを選ぶという貪欲アルゴリズムで解いた。あの貪欲アルゴリズムは、例えばユーロと同じ金額の配分であればうまくいくが、一般的にあのアルゴリズムは最適解を必ず生成するとは限らない。

ここでは、あらゆるコイン群に対しうまく動作する、動的計画法を使った効率的な解法を考えよう。動的計画法のアルゴリズムは、総当たりのように全ての組み合わせを探索する再帰関数に基づく。しかし動的計画法のアルゴリズムでは、メモ化により同じ部分問題を繰り返し計算することを避けることにより効率化する。

漸化式の作成

動的計画法のアイデアは、問題を再帰的に定式化し、より小さな問題の解を用いて元の問題の解を得ることである。コインの問題では、最初の問題はこのようなになる: 合計金額を x とする必要なコインの最小枚数は?

$\text{solve}(x)$ を合計金額 x を満たすのに必要なコインの最小枚数とする。この関数の値は、個々のコインの額による。例えば、コインの額が $\text{coins} = \{1, 3, 4\}$ である場合、この関数の値は以下の通りとなる:

$\text{solve}(0)$	$=$	0
$\text{solve}(1)$	$=$	1
$\text{solve}(2)$	$=$	2
$\text{solve}(3)$	$=$	1
$\text{solve}(4)$	$=$	1
$\text{solve}(5)$	$=$	2
$\text{solve}(6)$	$=$	2
$\text{solve}(7)$	$=$	2
$\text{solve}(8)$	$=$	2
$\text{solve}(9)$	$=$	3
$\text{solve}(10)$	$=$	3

例えば $\text{solve}(10) = 3$ である。なぜなら、合計金額を 10 とするには 3 枚のコインがあればよく、最適解で $3 + 3 + 4 = 10$ が存在するためである。

solve の重要な性質は、この値はより小さな値から再帰的に計算できる点である。ここで合計を計算するにあたり最初に選ぶコインに着目してみる。例えば、先ほどの例では最初に選ぶコインは 1, 3, 4 のいずれかである。もし最初に 1 を選ぶと、残りの問題は合計 9 を達成する最小コイン枚数を選ぶ問題となる。これは元の問題の部分問題となる。もちろん、同様の議論は 3, 4 のコインにも成り立つ。よって、最小コイン枚数に関する以下の漸化式が利用できる。

$$\begin{aligned}\text{solve}(x) = \min(&\text{solve}(x-1)+1, \\ &\text{solve}(x-3)+1, \\ &\text{solve}(x-4)+1).\end{aligned}$$

この漸化式の始まりは $\text{solve}(0) = 0$ である。合計金額 0 を満たすのは明らかにコイン 0 枚であるためである。よって、例えば以下のように計算できる。

$$\text{solve}(10) = \text{solve}(7) + 1 = \text{solve}(4) + 2 = \text{solve}(0) + 3 = 3.$$

ここから、一般的なコインの組み合わせに関して、同様に合計金額 x を満たす最小コイン枚数を求める漸化式が構成できる:

$$\text{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \text{coins}} \text{solve}(x-c) + 1 & x > 0 \end{cases}$$

まず $x < 0$ の場合、値は ∞ である。なぜなら合計金額が負となる選び方は不可能であるためである。 $x = 0$ の場合、値は 0 となる。これは合計金額が 0

となるのにコインは1枚も不要なためである。最後に $x > 0$ のとき、変数 c は1枚目のコインとして選べる全てのコインに関し探索する。

一度問題を解くための漸化式が構築できれば、それを C++ で直接実装できる (定数 INF は無限大を意味するものとする):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

またこの関数は効率的でない。この式では、解を得るたびに必要な計算回数が指数関数的に増加する場合があるためである。次に、メモ化と呼ぶ手法を使いこの関数を効率的にする方法を見ていこう。

メモ化の利用

動的計画法では、再帰的な関数を効率的に計算するためメモ化(**memoization**)を用いる。これは、関数の値を計算後配列に覚えておき、同じ入力に対して再度同じ関数が呼び出されたらその値を取り出して再計算を避ける手法である。

この問題では、以下の配列を使うものとする:

```
bool ready[N];
int value[N];
```

`ready[x]` は `solve(x)` の値が計算済みかどうかを示す。そして計算済みであれば、`value[x]` にその値を格納する。定数 N は必要な値がすべて配列に格納しきれるように選ぶ。

こうすると、先の関数は以下の通り効率的に実装できる:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}
```

この関数では、 $x < 0$ と $x = 0$ の場合は以前の解法と同じである。そうでない場合、関数では `ready[x]` をチェックして `solve(x)` の解がすでに `value[x]` に格納済みか判定する。格納済みであればその値を返して関数は終了し、そうでなければ関数は `solve(x)` の値を計算し、`value[x]` に格納しておく。

この関数は、各入力 x に対し一度ずつしか再帰的な計算を行わないため、効率的に動作する。`solve(x)` の値が `value[x]` に格納された後は、同じ入力 x に対しそれを取り出すだけで済むためである。このアルゴリズムは、合計金額を n 、コインの種類を k とすると時間計算量 $O(nk)$ となる。

なお、入力 $0 \dots n$ に対する `solve` の値を、再帰を使わず単純なループで繰り返し構築していくこともできる:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

実際、多くの競技プログラマはこちらの実装の方が短く定数倍高速なためこちらの書き方を好む。以降、本書で例を示す際、こちらのループによる実装も使用していく。しかし、動的計画法は再帰的な関数の観点で解を考えた方が易しい場合もある。

解の構築

時々、最適値を求めるだけでなく最適値の構成例を求める問題に出会うことがある。コインの問題では、例えば別の配列に求める総額を達成するために最初の 1 枚で選ぶコインを代入しておくことでこのような問題に対応することができる:

```
int first[N];
```

この配列を使い、先ほどのアルゴリズムをこのように書き換える:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}
```

その後、以下のコードは合計金額を n にするようなコインの構成を出力する:

```
while (n > 0) {  
    cout << first[n] << "\n";  
    n -= first[n];  
}
```

解の数え上げ

今度はコインの問題の別バージョンとして、合計金額 x とするコインの選び方が何通りあるか数える問題を考えよう。

例えば $\text{coins} = \{1, 3, 4\}$ かつ $x = 5$ であれば、計 6 通りの選び方がある;

- $1+1+1+1+1$
- $1+1+3$
- $1+3+1$
- $3+1+1$
- $1+4$
- $4+1$

こちらも、再帰的に解くことができる。 $\text{solve}(x)$ を合計金額を x とするコインの選び方の数とする。例えば $\text{coins} = \{1, 3, 4\}$ であれば $\text{solve}(5) = 6$ である。この関数に関して、以下のように再帰的に値を計算できる:

$$\begin{aligned}\text{solve}(x) = & \text{solve}(x-1) + \\ & \text{solve}(x-3) + \\ & \text{solve}(x-4).\end{aligned}$$

コインの種類に関し、一般形はこのように表せる:

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x-c) & x > 0 \end{cases}$$

もし $x < 0$ ならば、明らかにそのような構成は存在しないので値は 0 となる。 $x = 0$ であれば、コインを何も選ばないという組み合わせのみ存在するので値は 1 となる。それ以外の場合、 coins に含まれる全ての c に対し、 $\text{solve}(x-c)$ の値を合計したものが値となる。

$\text{count}[x]$ を $\text{solve}(x)$ の値を $0 \leq x \leq n$ の範囲で格納する配列だとすると、この配列はこのように作ることができる:

```
count[0] = 1;  
for (int x = 1; x <= n; x++) {  
    for (auto c : coins) {  
        if (x-c >= 0) {  
            count[x] += count[x-c];  
        }  
    }  
}
```

```
}
```

しばしば求める組み合わせの数が非常に大きくなるため、正確な値ではなく代わりにある m の剰余、例えば $m = 10^9 + 7$ の剰余を答えさせる問題がある。この場合、先ほどのコードにおいてすべての計算に m の剰余を求める式を加えるとよい。先ほどのコードであれば、下記のコードを:

```
count[x] %= m;
```

以下の行の後に加える:

```
count[x] += count[x-c];
```

ここまでで動的計画法の基本アイデアについて述べた。動的計画法は様々な状況において利用できるため、その可能性に触れられるいくつかの問題について触れていくことにする。

7.2 最長増加部分列


最初の問題は、 n 要素の配列における最長増加部分列 (**longest increasing subsequence**) を見つける問題である。最長増加部分列とは、元の数列の部分列のうち、選んだ要素を順に並べると手前の要素より後の要素の方が大きいという条件を満たす最長のものである。

例えば、以下の数列において:

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

最長増加部分列は 4 要素からなる:

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3



$\text{length}(k)$ を、 k 番目の要素を終端とする最長増加部分列の長さとする。 $\text{length}(k)$ の値を $0 \leq k \leq n-1$ 全てについて求めていくことにする。先ほどの例では、この関数の値は以下ようになる:

```
length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2
```

例えば $\text{length}(6)=4$ となるのは、6 番目の要素を終端とする最長増加部分列は 4 要素で構成されることを意味する。

$\text{length}(k)$ を求めるためには、 $i < k$ かつ $\text{array}[i] < \text{array}[k]$ となる i のうち、 $\text{length}(i)$ が最大のものを見つけられれば良い。そうすれば、 i 番目の要素を末尾とする最長部分増加列に k 番目の値を追加することで、 $\text{length}(k) = \text{length}(i) + 1$ が得られることがわかる。そのような i が 1 つも存在しない場合、 $\text{length}(k) = 1$ 、すなわち k 番目の要素だけで構成される部分列しか構成できない。

この関数の値は、より小さな引数の結果から計算できるため、動的計画法が適用できる。以下のコードでは、先ほどの関数の値は配列 `length` に格納される。

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k], length[i] + 1);
        }
    }
}
```

このコードは二重ループであり時間計算量 $O(n^2)$ で動く。しかしこの処理はより効率的に時間計算量 $O(n \log n)$ で処理できるアルゴリズムが存在する。あなたはそのような方法がわかるだろうか？

7.3 グリッド中のパス

次の問題は、 $n \times n$ のグリッドにおいて、左上角のマスから始めて、右または下に移動しながら右下角のマスに到達するパスを見つける問題である。各マスには正整数が書かれており、経路したマスに書かれた整数の総和が最大となるパスを求めることを考えよう。

以下の絵はあるグリッドにおける最適手を示す：

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

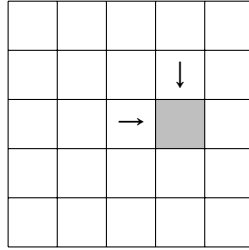
この図のパスにおける整数の和は 67 であり、これは最大である。

グリッドの行と列を 1 から n まで番号を振り、 $\text{value}[y][x]$ をマス (y, x) に書かれた値とする。 $\text{sum}(y, x)$ を左上マスからマス (y, x) に至るパスにおける経路したマスの整数値の総和の最大値とする。 $\text{sum}(n, n)$ は元の問題で求めたかった値に相当する。先ほどの例では $\text{sum}(5, 5) = 67$ となる。

この値は、このように再帰的に求めることができる：

$$\text{sum}(y, x) = \max(\text{sum}(y, x - 1), \text{sum}(y - 1, x)) + \text{value}[y][x]$$

この漸化式は、マス (y,x) を終端とするパスは、直前 $(y,x-1)$ か $(y-1,x)$ いずれかのマスから来るという考察により得られる:



そのため、和が最大となる方向を選ぶことができる。 $y=0$ かつ $x=0$ の場合、 $\text{sum}(y,x)=0$ と置くことにすると、この漸化式は $y=1$ や $x=1$ の場合にも適用できる。

この関数 sum は 2 つの引数を持ち、この動的計画法の配列は 2 次元である。例えば以下の配列を使うと:

```
int sum[N][N];
```

関数の値は以下のように求めることができる:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1], sum[y-1][x]) + value[y][x];
    }
}
```

このアルゴリズムの時間計算量は $O(n^2)$ となる。

7.4 ナップサック問題

ナップサック問題 (**knapsack**) は、物の集合が与えられて、ある特徴を満たす部分集合を求める類の問題を指す。ナップサック問題はしばしば動的計画法で解くことができる。

この節では、以下の問題に着目していく: n 個の重りの重さからなる配列 $[w_1, w_2, \dots, w_n]$ が与えられる。この重りの部分集合の総和として得られる重さをすべて列挙せよ。例えば重さが $[1, 3, 3, 5]$ の場合、以下の和を得ることができる。

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

この例では、 $0 \dots 12$ の範囲では 2 と 10 を除いた和が構成可能である。例えば部分集合として $[1, 3, 3]$ を選べば和として 7 を得られる。

この問題を解くため、部分問題として最初の k 個の重りだけを使って得られる重さの総和を考えよう。最初の k 個の重りの部分集合で重さの和を x にできるとき、 $\text{possible}(x, k) = \text{true}$ 、そうでないとき $\text{possible}(x, k) = \text{false}$ とする。この関数の値は、以下のように再帰的に求めることができる:

$$\text{possible}(x, k) = \text{possible}(x - w_k, k - 1) \vee \text{possible}(x, k - 1)$$

この式は、 k 個目の重りの重さ w_k を和に含める場合と含めない場合両方を考えることで導出できる。もし w_k を含めるなら、残りは $x - w_k$ の重さを最初の $k-1$ 個の重りで構成しなければならないし、 w_k を含めないなら x を最初の $k-1$ 個の重りで構成しなければならない。まず基本的なケースとして以下が成り立つ:

$$\text{possible}(x,0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

これは重りを 1 つも使わないなら、和は 0 にしかなりえないためである。

以下の表は、重さの配列が $[1,3,3,5]$ であるとき、対応する関数の値を列挙したものである (記号"X" は true であることを示す):

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

これらの値を計算し終わると、 $\text{possible}(x,n)$ が全ての重りを利用した場合、和を x にできるかどうかを示すことになる。

全ての重りの重さの総和を W とすると、動的計画法によりこのテーブルを埋めるための時間計算量は $O(nW)$ となる。対応するコードは以下の通り:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x-w[k] >= 0) possible[x][k] |= possible[x-w[k]][k-1];
        possible[x][k] |= possible[x][k-1];
    }
}
```

ここで、1次元配列 $\text{possible}[x]$ を使ったより良い実装法もある。この配列は、和が x となる部分集合を構築可能かどうかを示す。ここで重要なテクニックは、新しい重りに対する処理を、配列の右から左に更新していくことである:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

ここで紹介した基本的なアイデアは、多くのナップサック問題で利用できる。例えば、商品の重さと価値が与えられたとき、重さの総和に対し価値を最大化する場合などに適用できる。

7.5 編集距離

編集距離 (**edit distance**) またはレーベンスタイン距離 (**Levenshtein distance**)¹ は、ある文字列を別の文字列に変換するときに必要な編集回数の最小値である。1回の編集では、文字列に対し下記のいずれかの処理を行える:

- 文字を1つ挿入する (例: ABC → ABCA)
- 文字を1つ取り除く (例: ABC → AC)
- 文字を1つ変更する (例: ABC → ADC)

例えば、LOVE と MOVIE の編集距離は2である。まず最初の編集で文字の変更により LOVE → MOVE となり、続けて文字の挿入により MOVE → MOVIE とできる。この例では明らかに1回の編集では終えられないので、2が最小値であることがわかる。

以下、長さ n の文字列 x と、長さ m の文字列 y に対し、 x と y の編集距離を求めよう。問題を解くために、関数 $\text{distance}(a, b)$ を定義する。この関数は、接頭辞 $x[0 \dots a]$ と $y[0 \dots b]$ の編集距離を返すものとする。この関数を使えば、求める編集距離 $\text{distance}(n-1, m-1)$ となる。

この関数の値はこのように計算できる:

$$\begin{aligned} \text{distance}(a, b) = \min(&\text{distance}(a, b-1) + 1, \\ &\text{distance}(a-1, b) + 1, \\ &\text{distance}(a-1, b-1) + \text{cost}(a, b)). \end{aligned}$$

ここで、 $x[a] = y[b]$ なら $\text{cost}(a, b) = 0$ 、そうでなければ $\text{cost}(a, b) = 1$ とする。文字列 x に対し1文字加工するケースを考える:

- $\text{distance}(a, b-1)$: x の末尾に文字を加える
- $\text{distance}(a-1, b)$: x の末尾の文字を取り除く
- $\text{distance}(a-1, b-1)$: x の末尾の文字が一致する、または編集する

最初の2つのケースでは、1回の編集(挿入または削除)が必要である。最後のケースでは、 $x[a] = y[b]$ であれば最後の文字に関しては編集は不要である。そうでない場合、1回の編集(変更)が必要となる。

以下の表は、上記例における distance の値を示す:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

¹この名前は、この問題に取り組んだ V. I. Levenshtein に由来する [49].

表の右下角の値は、求める LOVE と MOVIE の編集距離である 2 を格納している。

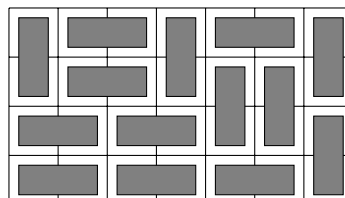
この表は、実際に編集距離を満たす編集の手順も示してくれる。先ほどの例では、以下のパスを考える：

		M	O	V	I	E	
		0	1	2	3	4	5
L		1	1	2	3	4	5
O		2	2	1	2	3	4
V		3	3	2	1	2	3
E		4	4	3	2	2	2

LOVE と MOVIE の末尾の文字は同じなので、これらの編集距離は LOV と MOVI の編集距離は同じである。次に、MOVI から I を取り除くため 1 回編集を行う。そして、LOV と MOV の編集距離は、同様に変更を行うためもう 1 増える。

7.6 敷き詰め方の数え上げ

動的計画法はしばしば単純な数の組み合わせより難しい解法になる場合もある。例として、 $n \times m$ のグリッドを、 1×2 または 2×1 の大きさのタイルで敷き詰める場合の敷き詰め方の数え上げ問題を考える。例えば、 4×7 のグリッドにおける 1 つの敷き詰め方はこの通りである：



そしてこの問題の総数は 781 である。

この問題は、行ごとに処理を進めていく動的計画法で解くことができる。各行を構成するマスに乗るタイルの形状を、 $\{ \sqcap, \sqcup, \sqsubset, \sqsupset \}$ からなる m 文字の文字列として表すことにしよう。例えば先ほどの例は、以下の文字列で表現できる：

- $\sqcap \sqsubset \sqcap \sqsubset \sqcap \sqcap$
- $\sqcup \sqsubset \sqcup \sqcap \sqcap \sqcup$
- $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$

$\text{count}(k, x)$ を、グリッドのうち $1 \dots k$ 行の範囲はすでにタイルが埋められており、かつ k 行目の状態が文字列 x に相当する埋め方をされている場合の組み合わせとする。ここで、行の状態は手前の行の状態をもとに構築できるので、動的計画法を活用することができる。

有効な解は、まず 1 行目が文字 \sqcup を含まず、 n 行目が文字 \sqcap を含まず、連続する行が互いに矛盾しない. ことである。例えば 2 つの行 $\sqcup \sqcup \sqcup \sqcup \sqcap \sqcap \sqcup$ と $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcap$ は矛盾しないが、 $\sqcap \sqcup \sqcup \sqcap \sqcup \sqcup \sqcap$ と $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ の行は矛盾する。

1 つの行の表現に m 個の文字が必要で、各文字 4 通りの組み合わせがあるので、行の状態は最大 4^m 通り考えられる。そのため、この解法の時間計算量は $O(n4^{2m})$ である。というのも、処理対象の行と手前の行はそれぞれ $O(4^m)$ 通りの状態を持つためである。この計算量は 4^{2m} の箇所が支配的なので、実際、グリッドを回転させ、 n より m が小さくなるようにするのは良いアイデアである。

行に対し、よりコンパクトな表現を用いることで、より効率化することもできる。前の行の情報として必要なのは、各マスが \sqcap かどうかだけである。よって、行の状態を \sqcap と \square の 2 種類の文字だけ考えるようにできる。(ここで \square は \sqcup, \sqcup, \sqcup のいずれかに相当する) この表現により、行の状態が 2^m 通りで表現できるため、全体の時間計算量は $O(n2^{2m})$ となる。

最後に付録として、この敷き詰め問題の驚くべき式を紹介する²:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot \left(\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right)$$

この式は、 $O(nm)$ で計算できるので非常に高速である。しかし計算過程で小数値の乗算を含むため、計算過程でどうやって中間状態を正確に保持するかという問題は残る。

²驚くべきことに、この式は偶然 2 人の研究チームが同じ年に発見した [43, 67]

Chapter 8

ならし解析

アルゴリズムの時間計算量は、しばしばアルゴリズムの構造から簡単に計算できる: 例えばアルゴリズムがいくつのループを含み、各ループが何回ループするかを数えればよい。しかし、しばしばそのような単純な分析では十分にアルゴリズムの真の姿をあぶりだせないことがある。

ならし解析（償却解析、**amortized analysis**）は時間計算量が変化するような演算を含むアルゴリズムを分析するのに利用できる。基本となるアイデアは、個々の処理に注目するのではなく、アルゴリズム実行の全体で各処理がどのように実行されるかから実行時間を推測することである。

8.1 尺取り法

尺取り法（**two pointers method**）では、配列全体を辿るのに 2 つのポインタを使う。両ポインタは一方向のみ動かすことで、アルゴリズムが効率的に動作することを保証する。ここから、尺取り法で解ける 2 つの問題について論じていこう。

部分列の総和

最初の例は、 n 要素の正整数の配列と求めるべき総和 x が与えられたとき、和が x となる部分列を探す問題である。

例えば、配列

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

は和が 8 となる部分列を含む。

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

この問題は尺取り法で $O(n)$ 時間で解くことができる。そのためのアイデアは、部分列の先頭と末尾のポインタを管理していくことである。処理 1 ループごとに、先頭のポインタを 1 つ後ろに動かし、その際末尾のポインタを部分列の総和が x 以下である限りできるだけ動かすことにする。もし途中で和がぴったり x になったら、解が見つかったことになる。

以下の例では、次の配列と求める総和 $x=8$ の場合を示す:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

最初の部分列は、値 1,3,2 を含み、その和は 6 である。

1	3	2	5	1	1	2	3
↑		↑					

次に、先頭のポインタを 1 つ右に動かす。末尾のポインタは、これ以上動かすと和が x を超えてしまうため動かさない。

1	3	2	5	1	1	2	3
	↑	↑					

再度先頭のポインタを右に動かし、今度は右のポインタを 2 つ右に動かす。そうすると得られる部分列の和は $2+5+1=8$ となり、求める部分列を見つけることができた。

1	3	2	5	1	1	2	3
		↑		↑			

このアルゴリズムの実行時間は末尾のポインタの移動回数に依存する。先頭ポインタを 1 回動かす間に末尾のポインタを何回動かすことになるかはわからないが、合計で $O(n)$ 回程度になることはわかる。末尾のポインタは配列を一方方向にしか動けないためである。

結局、先頭も末尾も両ポインタは $O(n)$ 回動くので、このアルゴリズムは全体で $O(n)$ 時間で動作する。

2 サム問題

尺取り法で解ける別の問題の例には 2 サム問題 (**2SUM problem**) がある。長さ n の配列と目標の和 x が与えられたとき、和が x となる 2 要素を探す問題である。

解法としては、まず数列を昇順ソートしよう。その後、2 つのポインタで配列を探索していく。左のポインタは先頭から末尾に 1 つずつ移動していく。右のポインタは逆に末尾から先頭に向けて移動していく。その際、両ポインタの指す要素の和が x 以下となるように移動する。もし和がちょうど x になるなら、求める解を見つけたことになる。

以下の配列について、目標の和を $x=12$ とした場合の例を考える：

1	4	5	6	7	9	9	10
---	---	---	---	---	---	---	----

初期状態で両ポインタの位置は下記のようにになる。両者の指す要素の和は $1+10=11$ であり x より小さい。

1	4	5	6	7	9	9	10
↑							↑

左のポインタを1つ右に動かす。右のポインタは対応して3つ左に動かすと、和は $4+7=11$ となる。

1	4	5	6	7	9	9	10
	↑			↑			

その後、左のポインタをもう一つ右に動かす。この場合右のポインタはこれ以上動かず、解として $5+7=12$ を得られる。

1	4	5	6	7	9	9	10
		↑		↑			

このアルゴリズムの実行時間は $O(n \log n)$ である。というのも、尺取り法部分は $O(n)$ 回のステップで処理が完了するが、最初のソートに $O(n \log n)$ 時間かかるためである。

なお、この問題は二分探索を使い $O(n \log n)$ 時間で解く別解もある。各要素について、それと対にして和が x となる要素を探す解法である。二分探索を使って探すようにすると、 n 回二分探索をするので全体で $O(\log n)$ の時間がかかる。

より難しい問題として、3つの要素の和が x となる要素を求める **3サム問題** がある。これは先ほどのアイデアを応用すると、 $O(n^2)$ 時間で解くことができる。¹ どのようにすればよいかわかるだろうか？

8.2 nearest smaller elements

ならし解析はしばしばデータ構造に対する処理数を概算するのにも用いる。アルゴリズムの個々のフェーズで実施する処理回数が不均等でも、全体で実施する処理量の上界が求められることはしばしばある。

例として、数値の配列が与えられたとき、**nearest smaller element**² を求めることを考える。これは、数列の各要素に対し、その手前の中で最も近くにある自身より小さい値を持つ要素のことである。（そのような要素が存在しない場合もある。）次に、スタックを使いこの問題を効率的に解く様子を見ていこう。

配列を左から右に辿っていき、同時に配列要素からなるスタックを管理して以降。配列の各位置において、スタックの最上位が自身の値より小さいか空になるまで最上位を **pop** することを繰り返していこう。そうすると、スタックに残った最上位の値が現在の要素に最も近いより小さい値となる。その後、自身をスタックに積む。

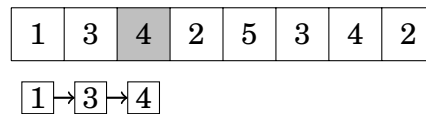
例えば以下の配列を考えよう：

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

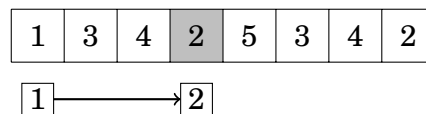
¹長い間、この問題は $O(n^2)$ 時間より短くは解けないと思われていた。しかし、2014年そうでないことが分かった。[30]

² 訳注：いい和訳が見つかりませんでした。

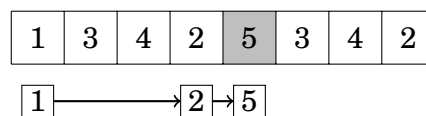
最初、要素 1,3,4 はこの順でスタックに積まれる。というのも、各要素は直前の値より大きいためである。その結果、4 に最も近いより小さい値は 3 であり、3 に最も近いより小さい値は 1 となる。



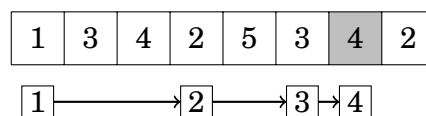
次の要素 2 は、スタックの上位 2 要素より小さい。そのため 3,4 はスタックから取り除かれ、2 がスタックに積まれる。よって 2 に最も近いより小さい値は 1 である：



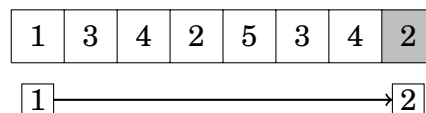
次の要素 5 は、スタックの最上位である 2 より大きい。そのため 5 がスタックに積まれる。5 に最も近いより小さい値は 2 となる：



その後、5 は取り除かれ 3,4 がスタックに積まれる：



最終的に 1 以外の要素は取り除かれ、最後の 2 がスタックに積まれる：



このアルゴリズムの効率性、スタックに対する処理の総回数による、もし現在の要素がスタックの最上位より大きければ、単に要素を最上位に積むだけでよく、効率的である。しかし、しばしばスタックに自身より大きな要素が積まれており、それらを取り除くのに時間がかかる場合がある。しかし、各要素は必ず 1 回だけスタックに積まれ、必ず 1 回だけスタックから取り除かれる。よって各要素のスタック処理回数は $O(1)$ 回であり、アルゴリズムは $O(N)$ 時間で動作する。

8.3 スライド最小値

スライドウィンドウ (**sliding window**) は配列中を左から右に移動する固定サイズの部分列である。このウィンドウの位置ごとに、ウィンドウ内の要素に関し何らかの情報を計算したいとする。この章では、スライド最小値

(**sliding window minimum**) に焦点を当てる。これはウィンドウ内の最小値を求めるものである。

スライド最小値は、先ほどの **nearest smaller element** と似たアイデアによって計算できる。各要素は手前の要素より大きく、かつ先頭要素がウィンドウ内の最小値と一致するようにしたキューを管理していくことを考える。ウィンドウを1つ動かすごとに、新たにウィンドウに入る要素より大きい要素が存在する限りそれらをキューの末尾から取り除いていく。そして先頭要素については、ウィンドウからはみ出てしまったものについては取り除く。最後に、新たにウィンドウに入る要素をキューの末尾に加える。

例として、以下の配列を考える。

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

スライドウィンドウのサイズを4としよう。最初の位置において、最小値は1である：

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 4 → 5

ウィンドウを1つ右に動かそう。新たな要素3はすでにキューにある4,5より小さいので、キューから4,5は取り除き、3を末尾に加える。最小値はまだ1のままである：

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 3

その後、再度ウィンドウを動かすと、最小値1はもはやウィンドウ内に無い。よって1をキューから取り除くと、最小値は3となる。また、新たな要素4がキューに追加される。

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

3 → 4

次の新たな要素1はキューのすべての値より小さいので、キューのすべての要素は取り除かれ、1だけが残る：

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1

ウィンドウが終端に到達すると、キューに要素2が追加されるが、ウィンドウ内の最小値は未だ1である。

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 2

配列の各要素がキューに出入りするのが高々1回なので、このアルゴリズムは $O(n)$ 時間で動作する。

Bibliography

- [1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [6] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [8] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).
- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [10] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [11] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [12] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, MIT Press, 2009 (3rd edition).

- [14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [15] K. Diks et al. *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, University of Warsaw, 2012.
- [16] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [17] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [19] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.
- [21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.
- [25] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [26] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [27] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [29] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>

- [30] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.
- [31] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [33] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.
- [34] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [35] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.
- [36] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [37] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [38] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [39] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [41] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [43] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [44] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [45] J. Kleinberg and É. Tardos. *Algorithm Design*, Pearson, 2005.
- [46] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).

- [47] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).
- [48] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [50] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [51] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [52] I. Parberry. An efficient algorithm for the Knight’s tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [53] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [54] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [55] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [56] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [57] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [58] S. S. Skiena. *The Algorithm Design Manual*, Springer, 2008 (2nd edition).
- [59] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003.
- [60] SZKOpU, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Staczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.
- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

