

競技プログラマー ハンドブック

Competitive Programmer's Handbook

原著: Antti Laaksonen

License: Creative Commons BY-NC-SA 4.0

2023 年 7 月 31 日

目次

Preface	ix
第 I 部 基本テクニック - Basic techniques	1
第 1 章 はじめに - Introduction	3
1.1 プログラミング言語	3
1.2 入出力	5
1.3 数字を扱う - Working with numbers	6
1.4 短く書く - Shortening code	9
1.5 数学 - Mathematics	11
1.6 コンテストや各種情報 - Contests and resources	16
第 2 章 時間計算量 - Time complexity	19
2.1 計算ルール - Calculation rules	19
2.2 時間計算量の種類 - Complexity classes	22
2.3 効率の見積もり - Estimating efficiency	23
2.4 部分配列の和の最大値 - Maximum subarray sum	24
第 3 章 ソートアルゴリズム - Sorting	29
3.1 ソートの理論 - Sorting theory	29
3.2 C++ のソート - Sorting in C++	34
3.3 二分探索 (バイナリサーチ) - Binary search	36
第 4 章 データ構造 - Data structures	41
4.1 動的配列 - Dynamic arrays	41
4.2 set - Set structures	43
4.3 map - Map structures	45
4.4 イテレータと範囲 - Iterators and ranges	46

4.5	その他の構造 - Other structures	48
4.6	ソートとの比較 - Comparison to sorting	52
第 5 章	全探索 - Complete search	55
5.1	部分集合を全生成する - Generating subsets	55
5.2	順列の生成 - Generating permutations	57
5.3	バックトラッキング - Backtracking	58
5.4	枝刈り - Pruning the search	60
5.5	半分全列挙 - Meet in the middle	63
第 6 章	貪欲アルゴリズム - Greedy algorithms	65
6.1	コイン問題 - Coin problem	65
6.2	スケジューリング問題 - Scheduling	67
6.3	締め切り付きタスク問題 - Tasks and deadlines	69
6.4	最小和問題 - Minimizing sums	70
6.5	データ圧縮 - Data compression	71
第 7 章	動的計画法 - Dynamic programming	75
7.1	コイン問題 - Coin problem	75
7.2	最長増加部分列 (LIS) - Longest increasing subsequence	81
7.3	グリッド上のパス - Paths in a grid	82
7.4	ナップザック問題 - Knapsack problems	83
7.5	編集距離 - Edit distance	85
7.6	タイルの数え上げ - Counting tilings	86
第 8 章	ならし解析 - Amortized analysis	89
8.1	2 ポインタ - Two pointers method	89
8.2	最も近い小さな要素 - Nearest smaller elements	91
8.3	最小スライディングウィンドウ - Sliding window minimum	93
第 9 章	区間クエリ - Range queries	95
9.1	静的なクエリ - Static array queries	96
9.2	BIT - Binary indexed tree	99
9.3	セグメントツリー - Segment tree	102
9.4	さらなるテクニック - Additional techniques	106
第 10 章	ビット操作 - Bit manipulation	109

10.1	ビット表現 - Bit representation	109
10.2	ビット操作 - Bit operations	110
10.3	集合の表現 - Representing sets	113
10.4	Bit optimizations	114
10.5	動的計画法へのビット演算の利用 - Dynamic programming	117
 第 II 部 グラフアルゴリズム - Graph algorithms		123
第 11 章	グラフの基礎知識 - Basics of graphs	125
11.1	グラフの用語 - Graph terminology	125
11.2	グラフの表現方法 - Graph representation	129
第 12 章	グラフ探索 - Graph traversal	135
12.1	深さ優先探索 - Depth-first search	135
12.2	幅優先探索 - Breadth-first search	137
12.3	応用 - Applications	139
第 13 章	最短経路	143
13.1	最短経路 (Bellman – Ford)	143
13.2	ダイクストラ法 - Dijkstra’s algorithm	146
13.3	ワーシャルフロイド法 - Floyd – Warshall algorithm	150
第 14 章	木のアルゴリズム - Tree algorithms	155
14.1	木の探索 - Graph Traversal	156
14.2	直径 - Diameter	157
14.3	全ノードからの最長経路 - All longest paths	159
14.4	二分木 - Binary trees	161
第 15 章	全域木 - Spanning trees	163
15.1	クラスカル法 - Kruskal’s algorithm	164
15.2	Union-find 構造 - Union-find structure	167
15.3	プリムのアルゴリズム - Prim’s algorithm	169
第 16 章	有向グラフ - Directed graphs	173
16.1	トポロジカルソート - Topological sorting	173
16.2	木 DP - Dynamic programming	175
16.3	サクセスパス - Successor paths	178

16.4	閉路検出 - Cycle detection	179
第 17 章	強連結 - Strong connectivity	183
17.1	Kosaraju's algorithm	184
17.2	2-SAT 問題 - 2SAT problem	186
第 18 章	木に対するクエリ - Tree queries	189
18.1	祖先の検索 - Finding ancestors	189
18.2	部分木とパス - Subtrees and paths	190
18.3	最小共通祖先 (LCA) - Lowest common ancestor	193
18.4	オフラインのアルゴリズム - Offline algorithms	196
第 19 章	経路と閉路 - Paths and circuits	201
19.1	オイラー路 - Eulerian paths	201
19.2	ハミルトン路 - Hamiltonian paths	205
19.3	デ・ブルーイェン配列 - De Bruijn sequences	207
19.4	ナイトツアー - Knight's tours	207
第 20 章	フローとカット - Flows and cuts	209
20.1	フォード・ファルカーソンのアルゴリズム - Ford – Fulkerson algorithm	211
20.2	素なパス - Disjoint paths	214
20.3	最大マッチング - Maximum matchings	216
20.4	辺被覆 - Path covers	219
第 III 部	発展的なテーマ - Advanced topics	223
第 21 章	整数論 - Number theory	225
21.1	素数と因数 - Primes and factors	225
21.2	mod の計算 - Modular arithmetic	230
21.3	方程式を解く - Solving equations	233
21.4	その他 - Other results	235
第 22 章	組合わせ論 - Combinatorics	237
22.1	二項係数 - Binomial coefficients	238
22.2	カタラン数 - Catalan numbers	241
22.3	包除原理 - Inclusion-exclusion	243

22.4	バーンサイドの補題 - Burnside's lemma	245
22.5	ケイリーの公式 - Cayley's formula	246
第 23 章	行列 - Matrices	249
23.1	行列と行列の演算 - Operations	250
23.2	線型回帰 - Linear recurrences	252
23.3	グラフと行列 - Graphs and matrices	254
第 24 章	確率 - Probability	257
24.1	確率の計算 - Calculation	257
24.2	事象 - Events	258
24.3	確率変数 - Random variables	261
24.4	マルコフ連鎖 - Markov chains	263
24.5	乱択	264
第 25 章	ゲーム理論	267
25.1	ゲームの状態 - Game states	267
25.2	Nim - Nim game	269
25.3	スプレイグ・グランディの定理 - Sprague – Grundy theorem	271
第 26 章	文字列アルゴリズム - String algorithms	275
26.1	文字列に関する用語 - String terminology	275
26.2	トライ構造 - Trie structure	276
26.3	文字列のハッシュ化 - String hashing	278
26.4	Z-アルゴリズム - Z-algorithm	281
第 27 章	平方根アルゴリズム - Square root algorithms	285
27.1	アルゴリズムの組み合わせ - Combining algorithms	286
27.2	整数のパーティション - Integer partitions	288
27.3	Mo のアルゴリズム - Mo's algorithm	290
第 28 章	発展的なセグメントツリー - Segment trees revisited	293
28.1	遅延伝搬 - Lazy propagation	294
28.2	動的セグメントツリー - Dynamic trees	297
28.3	データ構造 - Data structures	300
28.4	2次元セグメントツリー - Two-dimensionality	301
第 29 章	幾何学 - Geometry	303

29.1	複素数 - Complex numbers	304
29.2	点と線 - Points and lines	306
29.3	ポリゴンの面積 - Polygon area	309
29.4	距離関数 - Distance functions	311
第 30 章	掃引線アルゴリズム - Sweep line algorithms	315
30.1	交差点 - Intersection points	316
30.2	近接ペア問題 - Closest pair problem	317
30.3	凸包問題 - Convex hull problem	318
参考文献		321
Bibliography		321
索引		327

Preface

本書は、競技プログラミングの入門のために書かれました。プログラミングの基本をすでに知っていることが前提に書かれていますが競技プログラミング自身への予備知識は必要ありません。

想定読者としては特にアルゴリズムを学んで、国際情報学オリンピック (IOI) や国際大学対抗プログラミングコンテスト (ICPC) に参加したいと考えている学生を対象としています。それ以外にも競技プログラミングに興味のある人なら誰でも読むことができます。優れた競技プログラマになるには多くの時間を要しますが、さまざまなことを学べる機会でもあります。この本を読み、実際に問題を解き、コンテストに参加することに時間をかければ、アルゴリズムに対する理解が深まることは間違いありません。この本は継続的に開発されています。この本に対するフィードバックはいつでも ahslaaks@cs.helsinki.fi まで送ってください。(訳註: この本は 2022/04/20 時点の github 上の原稿を元に翻訳しています。誤訳や意識もあるので著者に連絡の際は原文を確認の上、連絡をしてください。)

(以下、原文)

The purpose of this book is to give you a thorough introduction to competitive programming. It is assumed that you already know the basics of programming, but no previous background in competitive programming is needed.

The book is especially intended for students who want to learn algorithms and possibly participate in the International Olympiad in Informatics (IOI) or in the International Collegiate Programming Contest (ICPC). Of course, the book is also suitable for anybody else interested in competitive programming.

It takes a long time to become a good competitive programmer, but it is also an opportunity to learn a lot. You can be sure that you will get a good general understanding of algorithms if you spend time reading the book, solving problems and taking part in contests.

The book is under continuous development. You can always send feedback on the book to ahslaaks@cs.helsinki.fi.

Helsinki, August 2019

Antti Laaksonen

オリジナル URL: <https://cses.fi/book/index.php>

翻訳版 github: <https://github.com/recuraki/cphb-ja>

第Ⅰ部

基本テクニック - Basic techniques

第 1 章

はじめに - Introduction

競技プログラミングとは大きく 2 つの段階に分かれます。アルゴリズムの設計とアルゴリズムの実装です。

アルゴリズムの設計は問題解決と数学的思考です。問題を分析し、想像力を活用して解決することが求められます。そのアルゴリズムは正確で効率的であることが求められ、多くの問題では効率的なアルゴリズムの考察が重要となります。

アルゴリズムの理論的な知識は競技プログラミングに取り組むにあたり重要な要素です。一般的な問題はよく知られたテクニックと新しい考察の組み合わせが必要となります。また、競技プログラミングに登場するテクニックはアルゴリズムの科学的な研究の基礎でもあります。

アルゴリズムの実装にはプログラミングスキルが重要です。競技プログラミングでは実装したアルゴリズムがテストケースによりテストすることで採点されるため、考察だけではなくて実装も正しくなければなりません。

コンテストでの良いコーディングスタイルとはシンプルで簡潔なものです。コンテストの時間は限られるため、プログラムは素早く書かなければなりません。通常のソフトウェア開発とは異なりプログラムは短いですし（どんなに長くても数百行程度）、コンテスト後にメンテナンスする必要はありません。

1.1 プログラミング言語

現在 (2018 年) 最も使われているプログラミング言語は C++, Python, Java です。Google Code Jam 2017 の上位 3000 人のうち 79% が C++, 16% が Python, 8% が Java を使用しています [29]. 複数の言語を使い分けている参加者もいます。

C++ が競技プログラミングに最適と考える人は多く、C++ はどのコンテストでも利用できます。C++ を使う利点は非常に効率的な言語であり標準ライブラリに

はデータ構造やアルゴリズムが豊富に揃っていることです。

一方、複数の言語を使いこなしてそれぞれの強みを理解するのも良いアプローチです。例えば、問題に (64bit や 128bit を超える) 大きな整数が必要な場合には Python は標準で大きな整数を扱えるため良い選択肢になります。ただし、多くの問題では特定のプログラミング言語を選択したことでアンフェアにならないようにされています。

本書で紹介するプログラムは C++11 に準拠しています。これは、多くのプログラミングコンテストで使うことができます。C++ には標準ライブラリのデータ構造やアルゴリズムが多く揃っており、あなたが C++ でプログラミングをしたことがなかったとすれば、今が勉強を始める良い機会です！

C++ のテンプレート

C++ での競技プログラミング用テンプレートを以下に示します。

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

using namespace std;

int main() {
    // solution comes here
}
```

最初の#include は g++ の機能で標準ライブラリを一括で読み込むことができます。多くのコードでよく使う iostream, vector や algorithm, を個別にインクルードせずに使えるようになります。

using 行は標準ライブラリの機能を使えるようにします。using がない場合は std::cout と書かないといけませんが、これがあることで cout だけで十分になります。

このコードは以下のようにコンパイルします。

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

このコマンドは test.cpp から test という実行形式のバイナリを作成します。(-std=c++11) は C++11 としてコンパイルすることを、(-O2) は最適化を行うことを、(-Wall) は全ての Warning を出力することを意味します。

1.2 入出力

ほとんどのコンテストでは標準入出力ストリームが用いられます。

C++ では標準入出力には、入力に `cin` が使われ、出力に `cout` が使われます。C の関数である `scanf` と `printf` も利用できます。

多くの問題で入力はスペースと改行で区切られた数字と文字列で構成されています。以下のように `cin` で入力ストリームから読み込むことができます。

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

`cin` は各要素の間に少なくとも 1 つのスペースか改行があることを前提に動作します。つまり、このコードは次の両方の入力を読み取ることができます。

```
123 456 monkey
```

```
123    456  
monkey
```

`cout` は次のように出力に使います。

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

入出力は時として実行時間のボトルネックになります。以下を用いることで効率的な入出力が可能です。

```
ios::sync_with_stdio(0);  
cin.tie(0);
```

改行には `"\n"` を用いると `endl` よりも高速に動作します。`endl` は出力 (`stdout`) の `flush` を行うからです。

`cin` と `cout` に代わり、C 言語では `scanf` と `printf` が存在します。通常、これらの関数は少し高速に動作しますが、使用するのが少し複雑になります。次のコードは入力から 2 つの整数を読み取ります。

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

次のように出力することができます。

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

問題によっては文字列を空白ごと読み込みたいことがあります。これには `getline` 関数が利用できます。

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

入力の文字列の数がわからない場合は次のように処理することもできます。

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

こうすることで入力に利用可能なデータがなくなるまで、入力から次々と要素を読み込みます。

いくつかのシステムではファイルの入出力を行う必要があります。この場合、以下のように書くことで標準入出力と同じように操作をすることができます。

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

この後、入力は"input.txt" から行われ、出力は"output.txt" に対して行われます。

1.3 数字を扱う - Working with numbers

整数 - Integers

最もよく使われる整数型は `int` という 32 ビットの整数型です。値の範囲は $-2^{31} \dots 2^{31} - 1$ です。競技プログラミングでは $-2 \cdot 10^9 \dots 2 \cdot 10^9$ と考えても良いでしょう。`int` 型では不十分な場合は、64 ビット型の `long long` を使用することができます。この型の値域は $-2^{63} \dots 2^{63} - 1$ で、同様におおよそ、 $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$ と考えられます。

次のコードでは `long long` 変数を定義しています。

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

数字の後の LL はこの数字は `long long` であるということです。

long long を使う時の注意は int との演算です。次の計算はうまくいきません。

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

変数 b は long long なのですが、a*a は int として演算されてから long long に格納されます。int で表現できる範囲を超えてしまうので誤った結果が格納されます。この例では a を long long に変更するか (long long)a*a というように結果をキャストするコードにすることで正しく動作します。

多くのコンテストの問題は、long long 型で十分なように設定されています。時折 g++ コンパイラが 128 ビットの __int128_t 型も提供していることを知っていると、値域が $-2^{127} \dots 2^{127} - 1$ 、つまり、 $-10^{38} \dots 10^{38}$ の範囲を扱えて便利です。ただし、この型がすべてのコンテストで利用できるわけではありません。(訳註: また多少低速になることもあります)

モジュロ演算 - Modular arithmetic

$x \bmod m$ というのは x を m で割った時のあまりといえます。例えば、 $17 \bmod 5 = 2$ で、 $17 = 3 \cdot 5 + 2$ と表せます。

答えの数が非常に大きくなる問題では「 m で割った数を答えよ」といった問題がよく出ます。例えば、 $10^9 + 7$ などが代表的です。これにより答えが非常に大きくても int や long long の値域で十分に扱えます。

ここで重要な性質は加算・減算・乗算において、演算の前に余りを取ることができることです。

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

つまり、演算のたびに余りを取れば数字がオーバーフローするほど大きくなりすぎることがありません。

n までの階乗である $n!$ を $\bmod m$ で求めてみましょう。

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

通常の余りは $0 \dots m-1$ で表現されます。ところが C++ やいくつかの言語ではマ

イナスの数に対する余りは負の数になってしまうため工夫が必要です。

あまりを求めてマイナスであった場合は m を加算することで解決できます。

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

これは負の可能性となることがある場合にのみ必要な処理です。

浮動小数点数 - Floating point numbers

競技プログラミングで多く用いられるのは 64-bit の浮動小数点型 `double` です。
g++ の拡張では 80-bit 浮動小数点型の `long double` を用いることができます。ほとんどのケースでは `double` で十分ですが、`long double` の方がより正確な値を表現できます。

答えに必要な精度は、問題文の中で示されています。桁を制限して答えを出力する簡単な方法は、`printf` 関数を使い、書式文字列で小数点以下の桁数を与える方法でしょう。

例えば、次のコードは、 x の値を小数点以下 9 桁で表示します。

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

浮動小数点数を使う際の留意点は浮動小数点数として正確に表現できない数値があつて丸め誤差が発生することです。

例えば、次のようなコードの結果は意外に思えるでしょう。

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.999999999999999988898
```

この回答は 1 ですが、丸め誤差のために x の値が 1 より少し小さくなってしまいました。この状態で `==` 演算子で浮動小数点数を比較すると、精度誤差のために本来等しいはずの値が等しくなくなる可能性があります。

浮動小数点数の比較に適した方法は 2 つの数値の差が ϵ (ϵ は小さな数) より小さければ等しいと仮定することです。

実際には、次のように数値を比較することができます ($\epsilon = 10^{-9}$ とした場合)。

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

尚、ある程度までの整数は正確に表現でき、`double` の場合は 2^{53} までの整数は正

しく表現できます。

1.4 短く書く - Shortening code

競技プログラミングでは、限られた時間で素早くプログラムを書くことが求められるので、短いコードが理想的です。そのため、競技プログラマは、データ型などのコードに短い名前を定義することがよくあります。

型の名前 - Type names

typedef を用いるとデータ型を短くできます。例えば、次のように long long という長い名前を ll とすることができます。

```
typedef long long ll;
```

このように宣言することで

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

というコードを次のように書けます。

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

typedef はもう少し複雑な型も表現できます。例えば次のように、整数の vector や、整数の pair を宣言できます。

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

マクロ - Macros

マクロ - macros も短く書くのに有効な手法です。マクロは、コード中の特定の文字列をコンパイル前に変更します。C++ では、マクロは#define キーワードを使って定義します。

例えば、次のようなマクロを定義することができます。

```
#define F first
```

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

これによって次のようなコードが、

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

次のようにかけるのです。

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

また、マクロは引数を持てるのでループなどを短く書くのに役立ちます。

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

このような定義をしておけば、

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

この記述が以下のように記載できます。

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

ただ、マクロは時折デバッグを難しくすることがあるので注意してください。よくある例を紹介します。

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

このような平方根を求めるマクロを書いたとします。

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

次のように展開されます。

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

このマクロは次のように書くべきでした。

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

すると

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

この処理は次のように展開されるので予想した通りに動きます。

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

1.5 数学 - Mathematics

競技プログラミングで数学は重要な役割を担っており、数学の能力がなければ競技用プログラマとして成功することは困難でしょう。この本の後半で必要となる重要な数学的概念と公式について説明します。

加算に関する公式 - Sum formulas

まず基本的な式を紹介します。

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

ここで、 k は正の整数であり、次数 $k+1$ の多項式となる閉形式を持ちます。他にも以下のようなものがあります。^{*1},

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

や

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

です。

等差数列 - arithmetic progression は隣り合う 2 つの要素の差が一定であるものです。

3, 7, 11, 15

^{*1} There is even a general formula for such sums, called **ファウルハーバーの公式 - Faulhaber's formula**, but it is too complex to be presented here.

この例の差は常に 4 です。等差数列の和は次の式で表されます。

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

a は初項、 b は最後の項、 n は項の数です。例えば次のようになります。

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

和が n 個の数からなり、各数の値が平均して $(a+b)/2$ のためこのような式になります。

等比数列 - geometric progression は隣り合う 2 つの値が決まった比により求められるものです。

$$3, 6, 12, 24$$

この例の差は常に 2 倍です。等比数列の和は次の式で表されます。

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

a は初項、 b は最後の項、 k は項の比です。

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

どのように求められるかをみていきます。

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

両辺に k をかけます。

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

これを整理して以下のように明らかです。

$$kS - S = bk - a$$

特殊なケースでは以下の公式が成り立ちます。

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

調和級数 - harmonic sum は次のような式のことです。

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

調和和の上限は $\log_2(n) + 1$ です。すなわち、 k が k を超えないような最も近い 2 の累乗になるように各項 $1/k$ を調整すればよいです。例えば、 $n = 6$ のとき、次のように見積もれます。

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

この時の上限は $\log_2(n) + 1$ ($1, 2 \cdot 1/2, 4 \cdot 1/4, \dots$) であり、それぞれは最大で 1 です。

集合論 - Set theory

集合 - **set** は要素のコレクションのことです。例えば集合

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

は 2, 4, 7 を要素として持ちます。特殊なシンボル \emptyset は空集合を意味します。 $|S|$ は集合に含まれる要素の数を示し、先ほどの例なら $|X| = 3$ です。

ある集合 S に要素 x が含まれている場合は $x \in S$ と示し、含まれていない場合は $x \notin S$ と示します。例をあげると次のようになります。

$$4 \in X \quad \text{であり} \quad 5 \notin X.$$

集合の演算により新しい集合を得ることができます。

- **集合積 - intersection** $A \cap B$ は A, B 両方に含まれている要素の集合です。
 $A = \{1, 2, 5\}, B = \{2, 4\}$ のとき、 $A \cap B = \{2\}$ です。
- **集合和 - union** $A \cup B$ は A, B いずれかに含まれている要素の集合です。
 $A = \{3, 7\}, B = \{2, 3, 8\}$ のとき、 $A \cup B = \{2, 3, 7, 8\}$ です。
- **補集合 - complement** \bar{A} は A に含まれていない要素です。補集合はこの集合を考える際の**全体集合 - universal set** に依存します。例えば、 $A = \{1, 2, 5, 7\}$ であるときに全体の集合が $\{1, 2, \dots, 10\}$ なのであれば $\bar{A} = \{3, 4, 6, 8, 9, 10\}$ です。
- **差集合 - difference** $A \setminus B = A \cap \bar{B}$ は A に含まれているが B に含まれていないものです。ここで、 B には A に含まれていない要素が含まれることもあります。 $A = \{2, 3, 7, 8\}$ で $B = \{3, 5, 8\}$ とするとき、 $A \setminus B = \{2, 7\}$ です。

A の全ての要素が S にも属する場合、 A は S の**部分集合 - subset** と呼び、 $A \subset S$ と表記します。集合 S は常に、空集合を含む $2^{|S|}$ 個の部分集合を持ちます。例えば、集合 $\{2, 4, 7\}$ の部分集合は次の通りになります。

$$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}, \{2, 4, 7\}.$$

次のような集合がよく使われます。 \mathbb{N} (自然数 - natural numbers), \mathbb{Z} (整数 - integers), \mathbb{Q} (有理数 - rational numbers), \mathbb{R} (実数 - real numbers). このうち \mathbb{N} は 0 を含むかは問題によって異なり、 $\mathbb{N} = \{0, 1, 2, \dots\}$ である場合もあれば、 $\mathbb{N} = \{1, 2, 3, \dots\}$ となる場合もあります。ことに注意します。

次のように集合を定義することもできます。

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

$f(n)$ は任意の関数です。これは、集合 S に含まれる要素 n を入力とした関数 $f(n)$ の結果全ての要素を含みます。

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

は全ての偶数を含むことになります。

論理式 - Logic

論理式の値は真 - **true** (1) か 偽 - **false** (0) であらわされます。代表的な論理式の記号を示します。 \neg (論理否定 - **negation**), \wedge (論理和 - **conjunction**), \vee (論理積 - **disjunction**), \Rightarrow (含意 - **implication**), \Leftrightarrow (対等 - **equivalence**).

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

$\neg A$ は A と反対の値です。 $A \wedge B$ は A, B の両方が真なら真です。 $A \vee B$ は A, B のどちらか (両方を含む) が真なら真です。 $A \Rightarrow B$ は、 A が真のとき B が真であるとき真で、 A が偽なら B はどちらでも良いです。 $A \Leftrightarrow B$ は A, B が両方同じ値のとき真です。

論理述語 - predicate は引数によって真または偽を返す式です。述語は通常、大文字で表記します。例えば、 x が素数のときだけ真になる述語 $P(x)$ が考えられます。この定義のときに $P(7)$ は真ですが $P(8)$ は偽を返します。

量化子 - quantifier は論理式と集合の要素を結びつけるものです。最も重要な量詞は \forall (すべての - **for all**) と \exists (ある - **there is**). です。例えば

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

は、集合の各要素 x に対して、 y が x よりも小さいような要素 y が集合に存在する、を表現できます。

このような表記法を用いると、さまざまな論理命題を表現することができます。

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

というのは、ある数 x が 1 より大きくて素数でないなら、1 より大きくてその積が x である数 a と b が存在するを表現できます。

関数 - Functions

$\lfloor x \rfloor$ は x を切り捨てる関数で、 $\lceil x \rceil$ は x を切り上げる関数です。

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

$\min(x_1, x_2, \dots, x_n)$ と、 $\max(x_1, x_2, \dots, x_n)$ はそれぞれ x_1, x_2, \dots, x_n の最大値と最小値を返す関数です。

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

$n!$ の表記は**階乗 - factorial** と呼ばれており次の式の通りです。

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

次のように再帰で表現もできます。

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

フィボナッチ数 - Fibonacci numbers は色々なところで登場する関数で次のように定義されます。

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

この数列は次のようになります。

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

フィボナッチ数は閉形式が存在します。これは **ビネの公式 - Binet's formula** と呼ばれています。

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

対数 - Logarithms

対数 - logarithm とはある x に対して $\log_k(x)$ で示されるもので、 k は底と呼ばれます。定義は $\log_k(x) = a$ のとき、 $k^a = x$ です。

対数の便利な性質は $\log_k(x)$ が x を k で 1 を下回らないまで割れる回数を表すことです。例えば $\log_2(32) = 5$ は 2 で 5 回割ることが出来ます。

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

アルゴリズムの解析には対数がよく使われます。多くの効率的なアルゴリズムは各ステップで何かを半分にするためです。そのため、対数を用いてそのようなアルゴリズムの効率を推定することができます。

対数の公式を見ていきます。

$$\log_k(ab) = \log_k(a) + \log_k(b),$$

$$\log_k(x^n) = n \cdot \log_k(x).$$

$$\log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b).$$

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)},$$

最後の式は任意の底への対数を計算することを可能にします。

自然対数 - natural logarithm $\ln(x)$ とは、 $e \approx 2.71828$ を底とする対数です。

対数のもう一つの性質は、底が b の整数 x の桁数は $\lfloor \log_b(x) + 1 \rfloor$ であることです (訳注: b 進数、と考えてよいです)。例えば、123 を底 2 で表すと 1111011 となり、 $\lfloor \log_2(123) + 1 \rfloor = 7$ となります。

1.6 コンテストや各種情報 - Contests and resources

IOI

国際情報オリンピック (IOI) は、毎年開催される中高生を対象としたプログラミングコンテストです。各国は 4 人の学生からなるチームを派遣でき、例年 80 カ国から約 300 名の参加者があります。

IOI は 5 時間に及ぶ 2 つのコンテストで構成されています。両コンテストとも参加者は様々な難易度の 3 つのアルゴリズム課題を解くよう求められます。タスクはサブタスクに分けられ、それぞれにスコアが設定されています。また、チームに分かれてはいますが個人ごとに競うことになります。

IOI のシラバス [41] は、IOI タスクに登場する可能性のあるトピックを掲載されており、この本ではこのうちのほとんどのトピックを掲載しています。

IOI への参加者は、国内コンテストを通じて選出されます。IOI の前には、Baltic Olympiad in Informatics (BOI)、the Central European Olympiad in Informatics (CEOI)、the Asia-Pacific Informatics Olympiad (APIO) などが開催されています。Croatian Open Competition in Informatics [11] や、the USA Computing Olympiad [68]. など、将来の IOI 参加者のためにオンラインコンテストを開催している国もあります。さらに、ポーランドのコンテストで出された問題の大規模なコレクションがオンラインで利用可能です [60]。

ICPC

国際大学対抗プログラミングコンテスト (ICPC) は、毎年開催される大学生を対象としたプログラミングコンテストです。1 チームは 3 名で構成され、IOI とは異なり各チームに 1 台しかないコンピュータを使い競います。

ICPC はいくつかのステージで構成され、勝ち進んだチームがワールドファイナルに招待されます。コンテストの参加者は数万人ですが、決勝の枠は限られています^{*2}。決勝戦に進出するだけでも大きな意味を持ちます。

ICPC コンテストでは、各チームが 5 時間の持ち時間で 10 問程度のアルゴリズムの問題を解き、すべてのテストケースを効率的に解くことができた場合のみ、その解答が認められます。コンテスト中は、他のチームの結果を見ることができません。ただし、最後の 1 時間はスコアボードがフリーズしてしまい、最後の投稿の結果を見ることができません。

ICPC で出題される可能性のあるテーマは、IOI のようにあまり特定されていません。ICPC ではより多くの知識、特に数学的スキルが必要とされるとされています。

Online contests

誰でも参加できるオンラインコンテストは数多くあります。最も活発なコンテストサイトは Codeforces で、毎週のようにコンテストを開催しています。Codeforces では、参加者を 2 つの部門に分け、初心者は Div2、経験豊富なプログラマは Div1 で競います。その他のコンテストサイトには、AtCoder、CS Academy、HackerRank、Topcoder があります。

いくつかの企業には、オンラインでコンテストを開催し、オンサイト（現地）で決勝を行うところもあります。Facebook Hacker Cup、Google Code Jam、Yandex.Algorithm などがその例です。もちろん、企業はこうしたコンテストを採用活動にも活用しています。コンテストで好成績を収めることは、自分のスキルを証明する良い方法でもあります。

書籍 - Books

競技プログラミングやアルゴリズムによる問題解決に焦点を当てた書籍は、(本書以外にも) すでにいくつかあります。

- S. S. Skiena and M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]

^{*2} The exact number of final slots varies from year to year; in 2017, there were 133 final slots.

- 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]

最初の 2 冊は初心者向けで、最後の 1 冊は上級者向けの内容です。

もちろん、一般的なアルゴリズムの本も、競技志向のプログラマーには良いです。人気のある本には、以下のようなものがあります。

- 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]

第 2 章

時間計算量 - Time complexity

競技プログラミングでは、アルゴリズムの効率性がとても大切です。動作時間が長くて良いのであればアルゴリズムを設計するのは簡単なことが多いですが、速いアルゴリズムを閃くことが大切になります。コンテストではアルゴリズムが遅すぎると、部分点しか取れなかったり全く点が取れなかったりしてしまいます。

アルゴリズムの時間計算量とは、ある入力の変数に対してそのアルゴリズムがどれだけの時間を使うかを推定するものです。つまり、時間の効率を入力変数のサイズをパラメータして関数で表現し、アルゴリズムが十分に速いかどうかを実装せずに検討することもできます。

2.1 計算ルール - Calculation rules

時間計算量は $O(\dots)$ という式で表され、3 つの点はなんらかの関数を示します。通常、変数 n は入力の大きさを表します。例えば、入力が数値の配列の場合 n は配列の大きさで、入力が文字列の場合は n は文字列の長さとなります。

ループ - Loops

アルゴリズムが遅くなる最も多い原因は、入力を処理する多くのループを含んでいることです。アルゴリズムに含まれるネストされたループが多ければ多いほど、プログラムの動作は遅くなります。もし、各ループが n 個の要素を処理する k 個のループがあるならその計算量は $O(n^k)$ になります。

例えば、以下のコードの時間計算量は $O(n)$ です。

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

次のコードの計算量は $O(n^2)$ になります。

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

オーダーの大きさ - Order of magnitude

時間計算量は、コードが実行された正確な回数を求めるものでなく計算量の大きさを示すだけです。以下の例ではループ内のコードは $3n$ 回、 $n+5$ 回、 $\lfloor n/2 \rfloor$ 回実行されますが、それぞれのコードの時間計算量は $O(n)$ と表現されます。

```
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
}
```

また、次の例では時間計算量は $O(n^2)$ です。

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

フェーズ - Phases

アルゴリズムがいくつかの連続的なフェーズで構成されている時、全体の計算量は最も大きなフェーズの計算量となります。その最も大きなフェーズが全体の動作のボトルネックになるためです。

例えば、以下のコードは、3つのフェーズから構成されてて、 $O(n)$, $O(n^2)$, $O(n)$

からなります。この場合、時間計算量は $O(n^2)$ と表現されます。

```
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
}
```

複数の変数が存在する場合 - Several variables

時間計算量は、複数の異なるサイズの変数の処理に依存することがあります。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 - Recursion

再帰的関数の時間計算量は、関数が呼び出される回数と、1回の呼び出しの時間計算量に依存し、これらの値の積として表現されます。

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

この場合は $f(n)$ は n の関数呼び出しを行い、それぞれの関数の処理は $O(1)$ です。このため、全体の時間計算量は $O(n)$ となります。

他の例も見てください。

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

```

    g(n-1);
}

```

この関数呼び出しは、 $n = 1$ を除いて、他に 2 つの呼び出しを発生させます。 g が呼ばれた時の関数の呼び出し回数を考えてみます。

function call	number of calls
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	2^{n-1}

この場合は以下のような時間計算量で表します。

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

2.2 時間計算量の種類 - Complexity classes

以下に時間計算量でよくある表現を紹介します。

$O(1)$ 定数時間 - constant-time は入力サイズにかかわらず一定の時間となるアルゴリズムです。

$O(\log n)$ 対数アルゴリズム - logarithmic は各ステップで入力サイズが半分になるアルゴリズムです。このようなアルゴリズムの実行時間は対数的で $\log_2 n$ となります。 n を 2 で割って 1 になる回数繰り返されるためです。

$O(\sqrt{n})$ 平方根アルゴリズム - square root algorithm は $O(\log n)$ より遅いものの $O(n)$ より早いアルゴリズムです。平方根の特殊な性質は $\sqrt{n} = n/\sqrt{n}$ であることで、これは入力の真ん中、を示します。(TODO: 直訳すぎる? どういう意味だ?)

$O(n)$ 線形 - linear アルゴリズムは入力を一定回数通過します。通常、答えを報告する前に少なくとも一度は各入力要素にアクセスする必要があるため、これが最良の時間複雑性であることがほとんどです。

$O(n \log n)$ 効率的なソートアルゴリズムの時間複雑度は $O(n \log n)$ となるので、ソートを要するアルゴリズムの多くはこの計算量になります。もちろん、アルゴリズムが各操作に $O(\log n)$ の時間を要するデータ構造を使用しているとこの計算量になります。

$O(n^2)$ 二乗 - quadratic アルゴリズムはネストしたループでよく目にします。特

に全てのペアを考える際には $O(n^2)$ となります。

$O(n^3)$ **三乗 - cubic** アルゴリズムも三重ループでよくみられます。これは三つの数字の組を走査する時によく出現します。

$O(2^n)$ これは集合において全ての組み合わせを処理する時にみられます。例えば、 $\{1, 2, 3\}$ という入力に対して、 $\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$ となります。

$O(n!)$ このアルゴリズムは全ての並び替えを試行する時に出現します。 $\{1, 2, 3\}$ の順列組み合わせは、 $(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$ となります。

アルゴリズムの最大の時間計算量が $O(n^k)$ である時、**多項式 - polynomial** 時間です。 $O(2^n)$ と $O(n!)$ 以外の上記の計算量は全て多項式時間と言えます。定数 k は通常小さいことが多く、多項式時間であることはそのアルゴリズムが効率的であることを意味するケースが多いです。

本書で紹介するアルゴリズムのほとんどは多項式時間です。しかし、多項式アルゴリズムが知られていない、つまり、誰もその効率的な解き方を知らない重要な問題もたくさんあります。このような **NP 困難 - NP-hard** な問題は重要な問題群です*1。

2.3 効率の見積もり - Estimating efficiency

アルゴリズムの時間計算により、そのアルゴリズムが問題に対して十分に効率的であるかどうかを実装前に確認することができます。ここで気にすべきなのはコンピュータが 1 秒間に何億回もの演算を行えるという事実とどの程度の計算ならば十分なのかということです。

例えば、ある問題の制限時間が 1 秒で、入力サイズが $n = 10^5$ であるとします。時間計算量が $O(n^2)$ である場合、このアルゴリズムは約 $(10^5)^2 = 10^{10}$ の操作を行うことになるでしょう。これは少なくとも数十秒かかるはずなので、この問題を解くにはアルゴリズムが遅すぎる、というのがわかります。

入力サイズがわかるならば、その問題を解くアルゴリズムの必要な時間複雑度を推測してみることができます。次の表は、制限時間を 1 秒と仮定した場合の有用な推定値です。

*1 A classic book on the topic is M. R. Garey's and D. S. Johnson's *Computers and Intractability: A Guide to the Theory of NP-Completeness* [28].

入力サイズ	許容できる計算量
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

(訳註: これにより、入力から明らかにないアプローチを除外することができます) 例えば、入力サイズが $n = 10^5$ の場合、アルゴリズムの時間計算量は $O(n)$ または $O(n \log n)$ であることが予想されます、といったように、より悪い時間複雑性を持つアルゴリズムを検討するアプローチを除外するため、アルゴリズムの設計を容易にします。

ただ、時間の複雑さは定数要素を隠してしまうので、効率の推定値に過ぎないということに留意してください。例えば、 $O(n)$ 時間で実行されるアルゴリズムはその内部で $n/2$ や $5n$ の演算を行うかもしれません。この場合、全体の実行時間はかなり長くなります。

2.4 部分配列の和の最大値 - Maximum subarray sum

ある問題を解決するためのアルゴリズムが複数存在し、その時間的複雑さが異なるというのはよくあることです。ここでは、 $O(n^3)$ の解を持つ古典的な問題を取り上げ、より良いアルゴリズムを設計することによって、この問題を $O(n^2)$ 時間で、さらには $O(n)$ 時間で解くことができることを示します。

n 個の数値からなる配列が与えられたとき、最大の部分和、すなわち配列中の連続した数値の最大和を計算する問題です*2。この問題は配列中に負の数がある時に非常に興味深い問題となります。

まずは例を見ていきましょう。

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

この場合の最大の部分和は 10 です。

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

空の配列も部分和と捉えるので 0 が考えられる最小の値となることに注意して

*2 J. Bentley's book *Programming Pearls* [8] made the problem popular.

ください。

方法 1 - Algorithm 1

この問題を解決するシンプルな方法は、可能性のあるすべての部分配列を調べて値の合計を計算し、最大合計を計算して候補を探し出すことです。次のコードはこのアルゴリズムを実装したものです。

```
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";
```

a と b で部分配列の両端を持ち、その間の和を sum で計算します。best はこの候補と比較して答えとなる最大値を持つものです。

ループから推測できる通り、 $O(n^3)$ の計算量となります。

方法 2 - Algorithm 2

少し検討するとループを 1 つ省略できることがわかります。右端を動かす時に和を計算すればよいです。

```
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";
```

こうすると、 $O(n^2)$ になりました。

方法 3 - Algorithm 3

しかし、この問題は $O(n)$ 時間で解くことができます*³。配列の各位置について、その位置で終了する部分配列の最大和を計算します。問題の答えは、それらの最大値となります。

このために、位置 k で終わる最大和の部分配列を求める補題を考えましょう。2つの可能性があります。

1. k のみが含まれる
2. $k-1$ で終わる部分配列に加えて k が含まれる

総和が最大となる部分配列を求めたいので、後者の場合、位置 $k-1$ で終了する部分配列はそれまでの総和が最大となるはずです。したがって、左から右へ各終了位置の最大の部分配列和を計算すれば、この問題を効率的に解くことができるはずです。

```
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";
```

これはループが1つしかなく、 $O(n)$ で動作します。この問題はどんな数字があるかを一度は確認しないといけないので、考える最良の計算量と言えます。

効率性の比較 - Efficiency comparison

アルゴリズムが実際にどの程度効率的であるかを研究するのは興味深いポイントです。次の表は、最新のコンピュータで、 n の値を変えて上記のアルゴリズムを実行した場合の実行時間です。

各テストでは、入力ランダムに生成し、入力の読み取り時間は測定していません。

*³ In [8], this linear-time algorithm is attributed to J. B. Kadane, and the algorithm is sometimes called **Kadane** のアルゴリズム - **Kadane's algorithm**.

要素数 n	方法 1	方法 2	方法 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

この比較から、入力サイズが小さいときにはどのアルゴリズムも十分に動作するが、入力サイズが大きくなるとアルゴリズムの実行時間に顕著な差が生じることがわかります。方法 1 は $n = 10^4$ のときに遅くなり、方法 2 は $n = 10^5$ のときに遅くなりました。方法 3 だけが、最大の入力でも瞬時に処理することができました。

第 3 章

ソートアルゴリズム - Sorting

ソートアルゴリズム - **Sorting** は典型的なアルゴリズムの問題です。多くのアルゴリズムはソートを使用しています。要素がソートされた順番になっていれば、データ処理が容易になる問題が多いからです。

例えば、「配列の中に同じ要素が 2 つあるか」という問題はソートを使えば簡単に解くことができます。ソートされている配列に 2 つの等しい要素があれば隣り合っているのも簡単に見つけることができます。また、「配列の中で最も頻度の高い要素は何か」という問題も同様に解くことができます。

ソートには多くのアルゴリズムがあり、様々なアルゴリズムを検討する良い例にもなっています。一般的に用いられるソートアルゴリズムは $O(n \log n)$ 時間で動作し、ソートをサブルーチンとして使用する多くのアルゴリズムもこの時間複雑性を持ちます。

3.1 ソートの理論 - Sorting theory

ソートの最も基本的な問題は次の通りです。

n 個の要素を含む配列が与えられたとき、要素を昇順に並べ替えてください。

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

この配列を次のように操作します。

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

$O(n^2)$ アルゴリズム

配列をソートするための最も簡単なアルゴリズムは、 $O(n^2)$ 時間で動作します。このアルゴリズムはシンプルに記述でき 2 つの入れ子ループで構成されます。有名な $O(n^2)$ のソートアルゴリズムを紹介します。**バブルソート - bubble sort** は各要素をその名の通りバブル (浮き上げ) させます。

バブルソートは n 回のラウンドで構成されて各ラウンドでは配列の要素を繰り返し処理します。連続する 2 つの要素で順序が正しくないペアが見つかったとそれを交換します。このアルゴリズムは以下のように実装することができます。

```
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]);
        }
    }
}
```

最初のラウンドのあと、最大の値は正しい場所にあることが保証されます。そして、 k ラウンドの後には、上から k 個の要素は正しい位置にあることが保証されます。つまり、 n ラウンド後にはソートが完了します。

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

この時の最初のラウンドを見てみます。

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



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



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



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



転置 - Inversions

バブルソートは、配列中の連続した要素を常に入れ替えるソートアルゴリズムの一例です。このようなアルゴリズムの時間計算量は常に $O(n^2)$ となります。なぜなら、最悪の場合、配列をソートするために $O(n^2)$ のスワップが必要になるからです。

ソートアルゴリズムを解析する際に有用な概念に**転置 - inversion** があります。 $a < b$ かつ $\text{array}[a] > \text{array}[b]$ であるような要素の数を示します。つまり、ソートされている順番に対して誤っているペアの数ともいえます。

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

を考えると 3 つの転置があります。(6,3), (6,5), (9,8) の値のペアです。反転の数は、配列の並べ替えにどれだけの作業が必要かを示します。転置がないとき、配列は完全にソートされています。一方、配列の要素が逆順（訳註：つまり降順）の場合に転地の数は最大となります。

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

隣り合う順序を入れ替えると配列からちょうど 1 つの反転だけが削除されます。つまり、ソートアルゴリズムが隣り合う要素の入れ替えしかできない場合に各入れ替えは最大でも 1 つの反転しか取り除けないため、アルゴリズムは少なくとも $O(n^2)$ です。

$O(n \log n)$ アルゴリズム

連続した要素の入れ替えに限らないアルゴリズムを用いれば $O(n \log n)$ で効率的に配列をソートすることが可能です。そのようなアルゴリズムの 1 つが、再帰に基づくマージソート^{*1} です。

マージソートは部分配列 $\text{array}[a \dots b]$ に対して以下のような操作を行います。

1. $a = b$ ならソートされているので何もしない
2. 中間のインデックスを計算する $k = \lfloor (a + b)/2 \rfloor$
3. 再帰的に $\text{array}[a \dots k]$ をソートする
4. 再帰的に $\text{array}[k + 1 \dots b]$ をソートする
5. ソートされた $\text{array}[a \dots k]$ と $\text{array}[k + 1 \dots b]$ を **マージ - Merge** し、ソートされた配列 $\text{array}[a \dots b]$ を作る

^{*1} According to [47], merge sort was invented by J. von Neumann in 1945.

マージソートは各ステップで部分配列のサイズを半分にすることで効率的に動作するアルゴリズムです。再帰は $O(\log n)$ 段の階層まで行われ、それぞれの段では $O(n)$ の時間を要します。部分配列 $\text{array}[a \dots k]$ と $\text{array}[k+1 \dots b]$ はソートされているので、線形時間でマージすることが可能です。

例えば、次のような配列のソートを考えてみましょう。

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

まず、次のように分割されます。

1	3	6	2
---	---	---	---

8	2	5	9
---	---	---	---

それぞれを次のようにソートします。

1	2	3	6
---	---	---	---

2	5	8	9
---	---	---	---

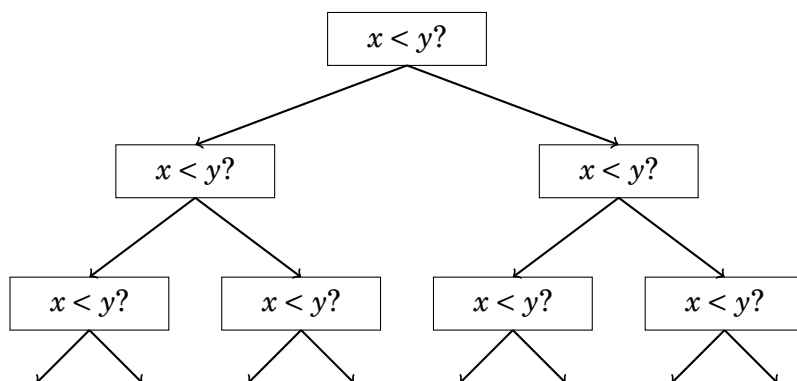
最後にその2つのソートされた配列を結合します。

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

ソート時間の限界 - Sorting lower bound

ソートの時間を $O(n \log n)$ より速くすることは可能でしょうか？ 配列要素の比較に基づくソートアルゴリズムに限定すると不可能であることがわかります。

ソートを2つの要素を比較するたびに配列全体の情報が得られる処理とみなし、時間計算量の限界を証明することができます。この処理を図にすると以下のような木が生成されます。



ここで、“ $x < y?$ ”とは、ある要素 x と y が比較されることを意味します。 $x < y$ ならば処理は左へ、そうでなければ右へ移動します。この処理の結果は配列の並べ替えの可能性を示しており、全部で $n!$ のノードからなります。これより、木の高さ

は少なくとも次の通り示されます。

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

最後の $n/2$ 個の要素を選び、各要素の値を $\log_2(n/2)$ に変更することと、この和の下界を得ることができます。

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

このように、 $n \log n$ がソートの限界であることが示されました。

カウントソート - Counting sort

ソートの計算量の限界が $n \log n$ ということを示しましたが、配列の比較が不要である場合はこの限りではありません。**カウントソート - counting sort** は各要素 c が $0 \dots c$ であることを仮定したソートで、 $O(n)$ で動作します。

このアルゴリズムは、元の配列を繰り返し処理し、各要素が配列中に何回現れるかを計算します。

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

この配列の例では出現回数の配列を次のように作ります。

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

この配列の 3 の位置の値は 2 で、要素 3 が元の配列に 2 回現れるを意味します。この構築には $O(n)$ の時間がかかり、各要素の出現回数を配列から取得できるため $O(n)$ でソート済み配列を作成することができます。したがって、ソート全体は $O(n)$ で動作します。カウントソートは非常に効率的なアルゴリズムですが定数 c が十分に小さい場合にのみ使用することができ、あまりに大きいとカウントの配列を持つことができません。

3.2 C++ のソート - Sorting in C++

ほぼ全てのプログラミング言語には優れた実装があるので、自作のソートアルゴリズムをコンテストで使用するのとはほとんど良い考えとは言えません。C++ の標準ライブラリには `sort` という関数があり、配列などのデータ構造のソートに簡単に利用することができます。

ライブラリ関数を使用することには多くの利点があります。まず、関数を実装する必要がないため、時間の節約になります。さらに、ライブラリの実装は正しく効率的です。自作のソート関数の方が良いということはまずあり得ないでしょう。

C++ の `sort` の使い方をみていきます。次のコードは配列をソートするものです。

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

この処理の後、配列自身が次のように変更されます。[2,3,3,4,5,5,8] となります。標準では昇順にソートされますが、つぎのようにすれば降順にできます。

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

`vector` でない配列も次のようにソート可能です。

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

文字列 `s` をソートすることもできます。

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

これは文字列の各文字をソートすることになり”monkey” は ”ekmnoy” とソートされます。

比較演算子 - Comparison operators

`sort` の関数には**比較演算子 - comparison operator** が定義されていないといけません。ソートを行う際に 2 つの要素の比較に常にこの演算子を使用されます。

C++ のほとんどのデータ型には比較演算子が組み込まれていて独自の比較演算関数を用意することなくソートできます。例えば、数値はその値に従ってソートされ、文字列はアルファベット順にソートされるというようになっています。

ペア - `pair` のソートは、主に最初の要素 (`first`) に従って並び替えられます。2 つのペアの第 1 要素が等しい場合は、それらは第 2 要素 (`second`) を用いてソートされます。

```
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());
```

この場合は (1,2), (1,5), (2,3) と並び替えられます。

タプル - `tuple` も似たようにソートされます。^{*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());
```

以下のようにソートされます。 (1,5,3), (2,1,3), (2,1,4)

^{*2} Note that in some older compilers, the function `make_tuple` has to be used to create a tuple instead of braces (for example, `make_tuple(2,1,4)` instead of `{2,1,4}`).

ユーザ定義構造体 - User-defined structs

ユーザ定義した構造体は比較演算子を定義しなければなりません。構造体で `operator<` を定義する必要がある、これは引数に同じ型の他の変数を取ります。小さいと判断する場合に `true` を、そうでない場合に `false` と返すようにします（訳註: 同じ場合は `false` を返すようにします）。

次の構造体 `P` には点の `x` 座標と `y` 座標が格納されています。比較演算子が定義されているため、この構造体はまず `x` で比較されて同値なら `y` で判定されます。

```
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 - Comparison functions

`sort` には外部の比較関数 - **comparison function** をコールバック関数として与えることもできます。次の例では比較関数 `comp` は第一に文字列長で比較し、同点の場合は辞書順で比較するソートを実現します。

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

この自作の比較関数を使って次のようにソートすることができます。

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

3.3 二分探索 (バイナリサーチ) - Binary search

配列の要素を検索する最も一般的な方法は、`for` ループを使用することでしょうたとえば、次のコードは、配列の要素 `x` を検索します。

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

```
}

```

このコードは最悪の場合で配列の全要素をチェックするので時間計算量は $O(n)$ となります。要素の順序が任意である場合は配列のどこを探せば要素 x が見つかるかという追加情報はないためこの方法は最良の方法です。

ただし、配列がソートされている状況では、より高速に探索を行うことが可能です。次の**二分探索 - binary search** アルゴリズムは、ソートされた配列の要素を $O(\log n)$ で効率的に探索します。

方法 1 - Method 1

二分探索の通常の実装方法は、辞書の中の単語を探すような作業といえます。この方法は配列のアクティブな領域を維持します。まず最初はすべての配列要素を含んでいます。その後、各ステップで領域のサイズを半分にします。

各ステップでは、アクティブな領域の中央の要素を確認します。中央の要素がターゲット要素であれば、探索は終了します。そうでない場合は、中央の要素の値に応じて、領域の左半分または右半分まで再帰的に探索をおこないます。

この実装は以下のようになります。

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

この実装ではアクティブな領域は $a \dots b$ で、初期値は $0 \dots n-1$ となります。各ステップで大きさを半分にするため、 $O(\log n)$ の計算量です。

方法 2 - Method 2

もう一つの方法は見る配列を効率的にする方法があります。このアイデアは大きな動きをだんだん小さくしていき、最後に目的の要素を見つけ出します。

探索は配列の左から右へ進めます。最初のジャンプの長さは $n/2$ です。各ステップでジャンプの長さは半分にし、次は $n/4$, 次に $n/8$, $n/16 \dots$ となり、最終的に長さは 1 になります。この結果、目的の要素が見つかったか、配列に現れないことが

分かります。

この実装は次のようになります。

```
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) {
    // x found at index k
}
```

b には今のジャンプの長さが含まれ、これを半分にしていくので計算量は $O(\log n)$ です。while は各ループに対して最大 2 回呼ばれます。

C++ の標準機能 - C++ functions

C++ 標準ライブラリには二分探索ベースの `log` で動作する探索の関数が用意されています。

- `lower_bound` x 以上となる最初の要素へのポインタを返します。
- `upper_bound` x よりも大きい最初の要素へのポインタを返します。
- `equal_range` はこの両方へのポインタを返します。

これらの関数は配列がソートされていることを前提として動作します。該当する要素がない場合、ポインタは配列の最後の要素の後を指します。次のコードは配列の中に値 x の要素があるかどうかを調べるものです。

```
auto k = lower_bound(array, array+n, x)-array;
if (k < n && array[k] == x) {
    // x found at index k
}
```

これらを活用すると次のコードは x の数をカウントすることができます。

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

C++ では `equal_range` という関数によりもっと簡潔に書けます。

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


最小の解 - Finding the smallest solution

注意するのは、二分探索というのは関数の値が変化する位置を見つけるという点です。ある問題に対して有効な解となる最小の値 k を求めたいとします。 x が有効な解であれば真を、そうでなければ偽を返す関数 $ok(x)$ を考えます。さらに $ok(x)$ は $x < k$ のとき偽、 $x \geq k$ の時に真を返すとします。

ここは次のように示せます。

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

このような関数がある場合に k の値は次のように二分探索で求めることができます。

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

この探索は、 $ok(x)$ が偽となる最大の値 x を探索します。次の値 $k = x + 1$ は、 $ok(x)$ が真となる最小の値といえます。ジャンプの長さ z の初期値は十分大きくないといけないことに注意してください。 $ok(z)$ が真であるような値にしてはならないということです。

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 の計算量に依存します。たとえば関数が $O(n)$ 時間で動作するのであれば全体の時間複雑度は $O(n \log z)$ となります。

最大の解の探索 - Finding the maximum value

二項探索は、まず増加して次に減少する関数の最大値を求める場合にも使用できます。次のような k を見つけたいとします。

- $x < k$ の時、 $f(x) < f(x+1)$
- $x \geq k$ の時、 $f(x) > f(x+1)$

これを二分探索で見つけるためのアイデアは $f(x) < f(x+1)$ となるような x の値を見つけることです。 $k = x+1$ であるため、 $f(x+1) > f(x+2)$ なることを意味します。次のように実装することができます。

```
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;
```

ここで、連続する値が同値となるケースは動作しないことに注意してください。
同値を引いた時にどちらに探索を続ければいいのかわからないためです。

第 4 章

データ構造 - Data structures

データ構造 -data structure は、コンピュータのメモリにデータを格納するための方法です。データ構造にはそれぞれ長所と短所があり、あるデータを格納する際に問題に適したデータ構造を選択できるかが大切になります。この判断に必要な知識は選択したデータ構造でどのような操作が効率的に行えるのかということです。

この章では、C++ 標準ライブラリの中で最も重要なデータ構造について紹介します。標準ライブラリを可能な限り使用することは、多くの時間を節約することができるのでベストな方法です。このドキュメントの後半では標準ライブラリでは利用できないより高度なデータ構造について学びます。

4.1 動的配列 - Dynamic arrays

動的配列 - dynamic array とは、プログラムの実行中にサイズを変更することができる配列のことです。C++ で最もよく使われる動的配列は `vector` で、普通の配列と同じように使うことができます。次のコードは、空の `vector` を作成し、そこに 3 つの要素を追加します。

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

これには配列のようにアクセスが出来ます。

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

size によって **vector** の長さを知ることが出来ます。これを利用して全ての配列の要素にアクセスすることができます。

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

また、for を使って次のようにアクセスをすることもできます。

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

back によって配列の最後の要素にアクセスすることができ、pop_back によって最後の要素を削除することが出来ます。

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

初期値を設定して生成することもできます。

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

次のようにして要素数を指定して作成したり、各要素の初期値を決めることが出来ます。

```
// size 10, initial value 0
vector<int> v(10);
```

```
// size 10, initial value 5
vector<int> v(10, 5);
```

vector の内部実装は、通常の配列を使用します。**vector** のサイズが大きくなり過ぎる、あるいは小さくなり過ぎるとき、新しい配列のメモリ確保して全要素を新しい配列に移動させます (訳註：この操作は $O(N)$ かかります)。ただし、このサイズ生成はあまり起こるものではないので push_back の平均的な時間複雑度は $O(1)$ と考えてよいです。

文字列型である `string` 構造体は動的な配列で、ほとんど `vector` のように使用することができます。`string` には他のデータ構造にはない特別な操作ができます。文字列は `+` 記号を使って結合することができます。関数 `substr(k,x)` は位置 k からの長さ x の部分文字列を返し、関数 `find(t)` はある文字列 t が最初に出現する位置 (訳註: あるいは存在しない) を探します。

次のコードでは、いくつかの文字列操作を紹介します。

```
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 set - Set structures

set は、要素の集合のデータ構造です。集合の基本操作は、要素の挿入、検索、削除からなります。

C++ の標準ライブラリには、2つの集合の実装があります。1つ目は `set` で、内部のデータは平衡二分木で格納され、操作は $O(\log n)$ 時間で行われます。もう一つは `unordered_set` 内部のデータをハッシュで格納し、操作は平均して $O(1)$ 時間で動作する。

どのセット実装を使うかは、好みの問題と特徴を考えて判断します。`set` の利点は、要素の順序を維持し、`unordered_set` では利用できない関数を提供されます。もちろん `unordered_set` の方が効率的な場合もある。

次のコードは、整数を含む集合を作成し、その操作の一部を示します。関数 `insert` は集合に要素を追加し、関数 `count` は集合内の要素の出現回数を返し、関数 `erase` は集合から要素を削除する関数です。

```
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
```

set の操作のいくつかは **vector** のように使うことができますが、[] 記法を使ってインデックスで指定した要素に直接アクセスすることはできません。次のコードは、セットを作成し、その中の要素数を表示し、すべての要素に処理をするものです。

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

集合の重要な特性は、そのすべての要素が**ユニーク - distinct** であることです。したがって、関数 `count` は常に 0 (要素ない) か 1 (要素ある) を返し、関数 `insert` は要素がすでに集合にある場合、それを追加することはありません。

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

C++ には、`multiset` と `unordered_multiset` という構造体もあり、これらは `set` と `unordered_set` と同様の機能しますが、同じ値の複数要素を格納することができます。たとえば、次のコードでは、5 という数字の 3 つのインスタンスがすべて `multiset` に追加されています。

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

`erase` は複数の要素があつたとしても全てを削除します。

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

1 つの要素だけを削除したいのなら次のようにイテレータでアクセスします。

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

4.3 map - Map structures

map はキーと値のペアで構成される一般的な配列のことである。通常の配列のキーは常に連続した整数 $0, 1, \dots, n-1$ (n は配列のサイズ) ですが、マップのキーは整数でない任意のデータ型とでき、連続した値である必要はありません。

C++ 標準ライブラリには、集合の実装に対応する 2 つのマップの実装がある。一つは map で平衡二分木に基づいており、要素へのアクセスに $O(\log n)$ 時間かかります。もう一つは unordered_map でハッシュを使用しており、平均で $O(1)$ で操作可能です。

次のコードは、キーが文字列、値が整数であるマップを作成します。

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

キーの値が要求されマップにその値が含まれていない場合、キーは自動的にデフォルト値でマップに追加されます（訳注：参照した時点で作成されることに注意します）。例えば、次のコードでは、値 0 を持つキー "aybabbtu" がマップに追加されます。

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

count はキーの存在を確認します。

```
if (m.count("aybabbtu")) {
    // key exists
}
```

次のコードはキーと値を列挙します。

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

4.4 イテレータと範囲 - Iterators and ranges

C++ 標準ライブラリの多くの関数は、**イテレータ - iterator** を使用して操作します。イテレータはデータ構造中の要素を指す変数です (訳註:C を知っている方はポインタと考えると良いでしょう)。

よく使われるイテレータを返す関数 `begin` と `end` は、データ構造内の全要素を含む範囲を示すために使われます。イテレータ `begin` はデータ構造の最初の要素を指し、`end` は最後の要素の後を指します。

```

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

```

`s.begin()` はデータ構造内の要素を指し、`s.end()` はデータ構造の外を指している、という非対称性に注意してください。つまり、イテレータで定義される範囲は片方が开区間 - *half-open* となっています。

範囲の利用 - Working with ranges

イテレータは、(訳註: 反復した処理に用いられることが多く) C++ の標準ライブラリ関数の中でデータ構造内の要素の範囲を与えられて使用されます。通常、データ構造内のすべての要素を処理したいので、イテレータ `begin` と `end` が操作の関数に与えられます。

次のコードは、`sort` 関数で **vector** をソートし、`reverse` 関数で要素の順序を反転させ、`random_shuffle` 関数で要素の順序をシャッフルします。

```

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

```

これらは通常の配列でも使え、この場合は配列のポインタを与えます。


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

イテレータの設定 - Set iterators

イテレータは、集合の要素にアクセスするためにもよく使われます。次のコードは、集合の最小の要素を指すイテレータ `it` を作成します。

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

次のように書くこともできます。

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

イテレータが指す要素には*記号でアクセスすることができます。たとえば、次のコードは、セットの最初の要素を表示します。

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

イテレータの移動は、演算子 `++` (次) および `--` (前) を用いて行うことができ、これはイテレータがセットの次の要素または前の要素に移動することを意味します。これらを使って、次のように全ての配列にアクセスすることもできます。

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

次のコードは配列の最後の要素を表示します。

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

関数 `find(x)` は、値が x である要素を指すイテレータを返す関数ですが集合に x が含まれない場合は、イテレータは `end` を示します。

```
auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}
```

関数 `lower_bound(x)` は、集合の中で値が x 以上である最小の要素へのイテレータを返し、関数 `upper_bound(x)` は、集合の中で値が x よりも大きい最小の要素へのイテレータを返します。これらの関数は、該当する要素がないならば、`end` を返します。また、要素の順序を保持しない `unordered_set` 構造体ではサポートされません。

例えば、次のコードは、 x に最も近い要素を見つけます。

```
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";
}
```

このコードでは、集合が空でないことを仮定し、イテレータ `it` を使用してすべての可能なケースを通過します。イテレータは値が少なくとも x である最小の要素を指します。もし `it` が `begin` に等しければ、対応する要素が x に最も近いです。もし `it` が `end` に等しければ、セットの中で最大の要素が x に最も近いです。もし前のケースがどれも成立しなければ、 x に最も近い要素は前後の要素のどちらかになります。

4.5 その他の構造 - Other structures

ビットセット - Bitset

ビットセット - **bitset** とは、各値が 0 または 1 である配列です。例えば、次のコードは 10 個の要素を含むビットセットを作成します。

```
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
```

ビットセットを使用する利点は、ビットセットの各要素は1ビットしか要さないもので少ないメモリしか必要としないことです。int 配列に n ビットを格納する場合、 $32n$ ビットのメモリを使用しますが、同じ長さのビットセットは n ビットのメモリしか必要としません。また、ビットセットの値はビット演算子を使って効率的に操作できるため、ビットセットを使ったアルゴリズムの最適化が可能です。

次のコードはビットセットを作成する別の方法を示しています。

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

関数 count は、ビットセット内の1の個数を返します。

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

また、次のようにビット演算を行うことができます。

```
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
```

デック - Deque

デック - **deque** は配列の両端で効率的にサイズを変更できる動的な配列です。vector と同様に deque は関数 push_back and pop_back を提供しますが vector では利用できない関数 push_front and pop_front も提供します。

```
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 よりも複雑なので遅いですが、要素の追加と削除は両端とも平均して $O(1)$ です。

スタック - Stack

スタック - stack は $O(1)$ 時間の操作を提供するデータ構造で、要素をトップに追加すること、トップから要素を削除することの 2 つの操作だけを提供します。つまり、スタックの先頭の要素にのみアクセスすることができる構造体です。

次のコードは、スタックを使用する方法を示しています。

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

Queue

キュー - queue も末尾に要素を追加する操作と、待ち行列の先頭の要素を削除する操作の 2 つの $O(1)$ の操作だけが用意されています。つまり、待ち行列の最初と最後の要素にのみアクセスすることができます。

次のコードは、キューを使用する方法を示しています。

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

優先度付きキュー - Priority queue

優先度付きキュー - priority queue は要素の集合を保持するデータ構造です。サポートされている操作は、挿入と、待ち行列の種類に応じて、最小または最大の要素の検索と削除です。挿入と削除には $O(\log n)$ の時間がかかり、取り出しには $O(1)$ の時間がかかります。。set でも、優先度付きキューでできるすべての操作が可能ですが、優先度付きキューは定数係数がより小さく操作できます。優先度付きキューはヒープ構造を用いて実装されますが、順序付きセットで用いられる平衡二分木よりもはるかに単純なものです。

C++ の優先度付きキューは、デフォルトは要素の降順でソートされ、キュー内の最大要素を見つけ、削除することが可能です。

```
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();
```

昇順（つまり小さい順）の優先度付きキューを作りたいならば次のようにします。

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

Policy-based data structures

g++ は、C++ 標準ライブラリに含まれないデータ構造もいくつかサポートしています。このような構造は、*policy-based* データ構造と呼ばれています。これらの構造体を使うには、次の行をコードに追加する必要があります。

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

これで set のように配列のようにインデックスを付けられるデータ構造 `indexed_set` を定義することができるようになりました。int を保持する構造は以下のように作ります。

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

次のように作ります。

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

```
s.insert(9);
```

この拡張 `set` はソートされた配列の要素にあるインデックスにアクセスできます。関数 `find_by_order` は、指定された位置の要素へのイテレータを返します。

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

`order_of_key` は与えた要素の `index` を返します。(TODO: これ本当か？ 2 ではない気がする)

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

その要素が集合に現れない場合、その要素が集合の中で持つであろう位置を求めます。

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

これらは対数時間で動作します。

4.6 ソートとの比較 - Comparison to sorting

データ構造とソートのどちらかを使って解ける問題は多いです。これらのアプローチの実際の効率に顕著な違いがでることもあり、それは時間的な複雑さに起因することがあります。

n 個の要素を含む 2 つのリスト A 、 B が与えられたときの問題を考えてみましょう。ここで両方に所属する要素数をカウントしたいとします。

例えば、以下の例を考えます。

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

この場合、2, 5, 9 が両方に含まれるので答えは 3 です。

愚直な比較を行うと $O(n^2)$ ですが、いくつかのアプローチを考えてみましょう。

方法 1: Algorithm 1

A の `set` をつくり、 B の各要素について A にも属するかどうかをチェックします。時間計算量は $O(n \log n)$ です。

方法 2: Algorithm 2

順序付きセットである必要がないので `set` の代わりに `unordered_set` を使うこともできます。基礎となるデータ構造を変更するだけなので非常に簡単な方法です。時間計算量は $O(n)$ になりました。

方法 3: Algorithm 3

ソートを使うことができます。まず A と B をソートします。この後、両方のリストを同時に反復処理し、共通の要素を見つけます。ソートの時間計算量は $O(n \log n)$ で残りのアルゴリズムも $O(n)$ 時間で動作するので、全体の時間計算量は $O(n \log n)$ となります。

効率性の比較 - Efficiency comparison

次の表は n が変化し、リストの要素が $1 \dots 10^9$ の間のランダムな整数である場合に、上記のアルゴリズムがどの程度効率的に動作するかを示します。

n	方法 1	方法 2	方法 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、使用する集合構造が異なることを除けば同じコードです。この問題では、この選択が実行時間に重要な影響を及ぼします。アルゴリズム 2 はアルゴリズム 1 より 4-5 倍速く動作します。

しかし、最も効率的な方法は、ソートを使用する方法 3 であることに注意してください。方法 2 に比べ、半分の時間しか使いません。興味深いことに、方法 1 と方法 3 の時間計算量はともに $O(n \log n)$ ですが、それにもかかわらず方法 3 は 10 倍も速いのです。

これは、ソートが方法 3 の冒頭で一度だけ行われるのに対して、残りの方法は線形時間で動作するためです。

一方、方法 1 はアルゴリズム全体の間、複雑なバランス二分木を維持するために毎回 $O(\log n)$ の動作を行うためです。

第 5 章

全探索 - Complete search

全探索 - Complete search は、ほとんどのアルゴリズムの問題を解決できる一般的な方法です。ブルートフォースとも呼ばれ、問題に対して可能な限りの解を生成します。全探索ですべての解を調べるのに十分な時間があれば、これは非常に良いアプローチです。

一般的に他の解法に比べて実装が簡単で正しい答えが得られるからです。完全探索が遅すぎる場合は貪欲アルゴリズムや動的計画法などの他の技法が必要になってくるでしょう。

5.1 部分集合を全生成する - Generating subsets

n 個の要素からなる集合のすべての部分集合を生成する問題を考えましょう。例えば $\{0, 1, 2\}$ の部分集合は、 $\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}$ and $\{0, 1, 2\}$ です。部分集合の生成には、再帰的探索を行う方法と、整数のビット表現を利用する方法の 2 つが一般的です。

Method 1

集合のすべての部分集合を調べる 1 つ目の方法は再帰です。次の関数 `search` は、集合 $\{0, 1, \dots, n-1\}$ の部分集合を生成します。この関数は、各集合の要素を含む **vector** の部分集合を保持します。この探索はパラメータとして `0` を与えることで開始されます。

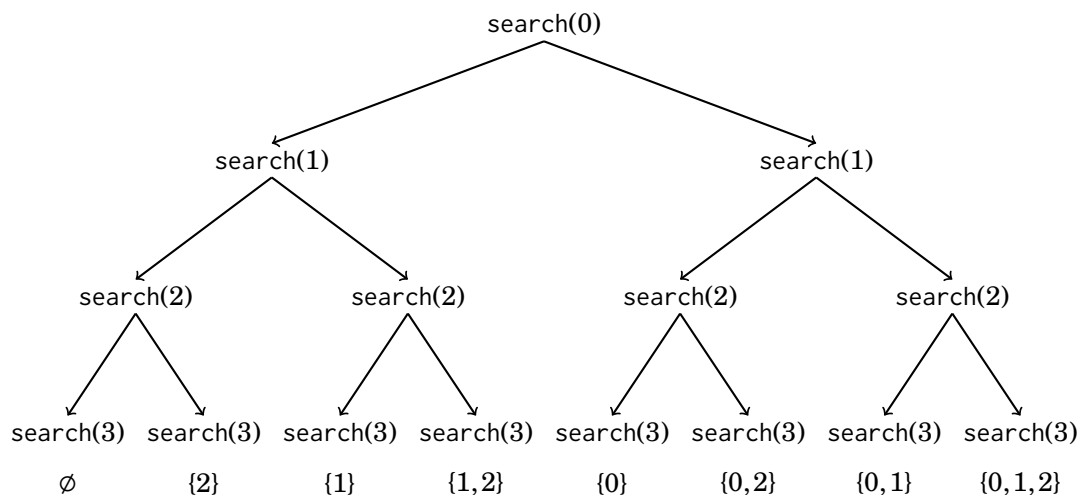
```
void search(int k) {
    if (k == n) {
        // process subset
    } else {
```

```

        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}

```

関数 `search` がパラメータ k で呼ばれると要素 k を部分集合に含めるかどうかを決定し、いずれの場合もパラメータ $k+1$ で自分自身を呼び出します。 $k=n$ の場合、関数はすべての要素が処理されて部分集合が生成されたとして処理をします。次のツリーは、 $n=3$ のときの関数呼び出しを示したものです。左の枝 (k は部分集合に含まれない) と右の枝 (k は部分集合に含まれる) のどちらが常に選ばれます。



Method 2

部分集合を生成するもう一つの方法は、ビット列を使う方法です。 n 個の要素からなる集合のそれぞれの部分集合は $0 \dots 2^n - 1$ の間の整数に対応する n ビット列として表現できます。1 のビットはど部分集合に含まれるかを示します。一般的な実装では最後のビットが要素 0 に対応し、2 番目のビットが要素 1 に対応し.. となります。例えば、25 のビット表現は 11001 で、部分集合 $\{0,3,4\}$ に対応します。

次のコードは、 n 個の要素を持つ集合の部分集合を調べます

```

for (int b = 0; b < (1<<n); b++) {
    // process subset
}

```

次のコードは、ビット列に対応する部分集合の要素を見つける方法です。各部分集合を処理するとき、コードはサブセットの要素を含む **vector** を返します。

```

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 - Generating permutations

n 個の要素からなる集合のすべての並べ換えを生成する問題を考えましょう。例えば、 $\{0, 1, 2\}$ の並べ換えは、 $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$, $(2, 1, 0)$ です。ここでも 2 つのアプローチがあります。再帰を使うか、順列を繰り返し見ていくかです。

Method 1

再帰のアプローチです。次の関数 `search` は、集合 $\{0, 1, \dots, n-1\}$ の並べ換えを列挙します。この関数は、並べ換えを含む `vector` を構築し、パラメータなしでこの関数を呼ぶと生成を開始します。

```

void search() {
    if (permutation.size() == n) {
        // process permutation
    } 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();
        }
    }
}

```

関数呼び出しのたびに新しい要素が `permutation` に追加されます。配列 `chosen` はどの要素がすでに入ったかを記録します。そして、`permutation` の長さが求めたいものと一致した時に、それを結果として処理します。

Method 2

もう一つの方法は、この順列を $\{0, 1, \dots, n-1\}$ から開始して順番に次の順列を求めていくことです。C++ の標準ライブラリにはこのための関数 `next_permutation` があり、次のように使うことができます。

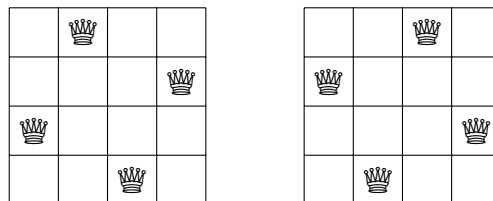
```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));
```

5.3 バックトラッキング - Backtracking

バックトラッキング - backtracking とは、空の解から始めて段階的に解を作っていく方法の総称です。探索は解がどのように構築されうるかを再帰で全探索していきます。

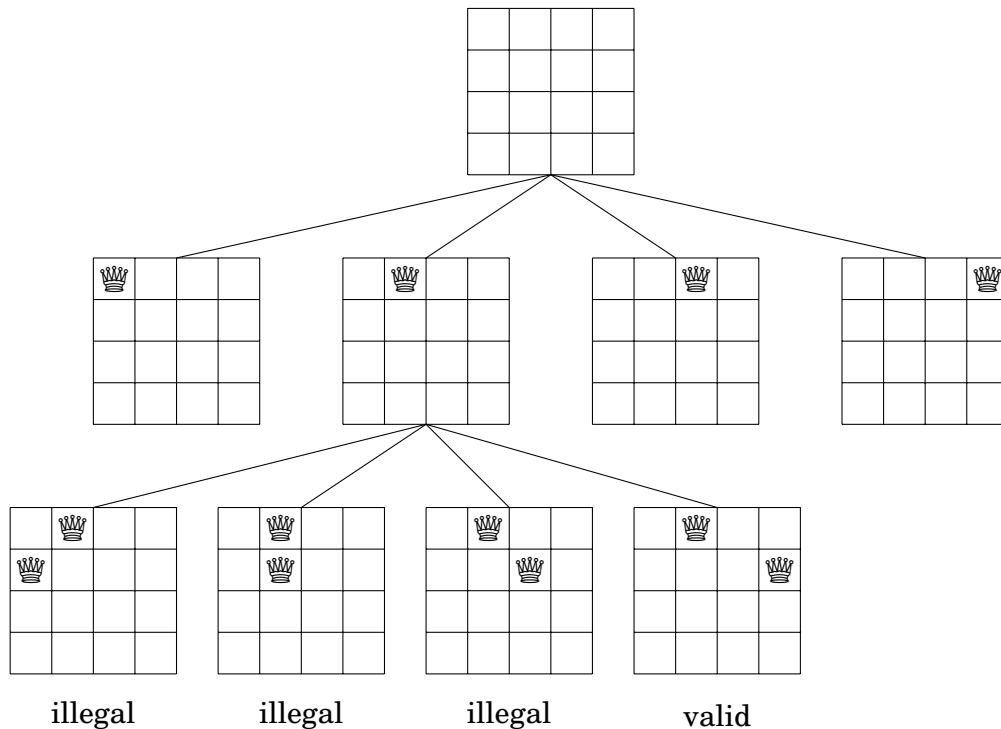
バックトラッキングを考えるうえで、 $n \times n$ のチェス盤の上に、2つの女王が互いに攻撃し合わないようにな個の女王を配置する方法の数を計算する問題を考えましょう。これは、クイーン問題として知られています。

$n = 4$ のときの解の例を2つ示します。



クイーン問題はバックトラックを使用して一列ずつクイーンを配置することで解くことができます。各ターンでそれまでに置かれたクイーンを攻撃するクイーンがないように、各列にちょうど1つずつクイーンを置いていきます。 n 個のクイーンがすべてボード上に配置されたときに解が得られます。

例えば、 $n = 4$ の場合、バックトラックアルゴリズムで生成される部分解は以下のようになります。



図の下に示した最初の3つはクイーン同士が攻撃しあっているので不正な回答です。4番目の構成は有効で、さらに2つのクイーンをボードに配置することで完全な解答に拡張することができます。なお、残りの2つのクイーンを配置する方法は1つだけです。これは次のように実装することができます。

```
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;
    }
}
```

`search(0)` を呼び出すことで探索が開始されます。盤の大きさは $n \times n$ であり、あり得る解の数を計算します。

このコードでは、盤面の行と列は 0 to $n-1$ までの番号が付けられているとします。関数 `search` がパラメータ y で呼ばれると、 y 行にクイーンを置ける場所を探し $y+1$ で自分自身を呼び出します。そして $y=n$ ならば解が見つかったことにな

り count を増加させます

配列 column はクイーンを含む列を、配列 diag1 および配列 diag2 は対角線を示します。すでにクイーンを含む列や対角線にさらにクイーンを追加することはできません。例えば、 4×4 のボードの列と対角線は次のように番号付けします。

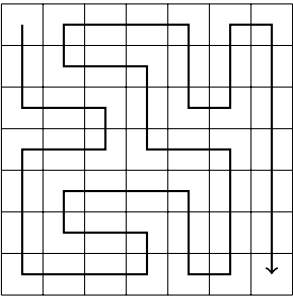
0	1	2	3	0	1	2	3	3	4	5	6
0	1	2	3	1	2	3	4	2	3	4	5
0	1	2	3	2	3	4	5	1	2	3	4
0	1	2	3	3	4	5	6	0	1	2	3
column				diag1				diag2			

$n \times n$ のチェス盤に n 個のクイーンを配置する方法の数を $q(n)$ とします。上記のバックトラックアルゴリズムにより、例えば、 $q(8) = 92$ となることがわかります。 n が大きくなると解の数は指数関数的に増えて探索はすぐに遅くなってしまいます。例えば、上記のアルゴリズムを使って $q(16) = 14772512$ を計算すると、最新のコンピュータでも 1 分以上かかります。^{*1}。

5.4 枝刈り - Pruning the search

バックトラックを探索木の枝刈りで最適化することができる場合があります。このアイデアは、アルゴリズムに”intelligence”を加えることで、部分解が完全解に拡張できない場合、できるだけ早くそれに気づきその先の探索を打ち切りますこのような最適化は、探索の効率に多大な影響を与える。

ここで、 $n \times n$ の格子において、左上から右下に向かう経路が各マスをちょうど 1 回ずつ訪れるような経路の数を計算する問題を考えてみましょう。例えば、 7×7 のマス目では、111712 本のパスが存在します。例として、そのうちの 1 本は次のようなものです。



^{*1} There is no known way to efficiently calculate larger values of $q(n)$. The current record is $q(27) = 234907967154122528$, calculated in 2016 [55].

7×7 のケースは難易度がほどほどに高く、我々のニーズに合っているのでこのケースを取り上げることにします。まず、素直なバックトラックアルゴリズムから始め、探索をどのように刈り込むことができるかの観察を用いて、段階的に最適化する方法を見ていきます。各最適化の後、アルゴリズムの実行時間と再帰呼び出しの数を測定し、各最適化がどのような影響を与えたのかを調べてみましょう。

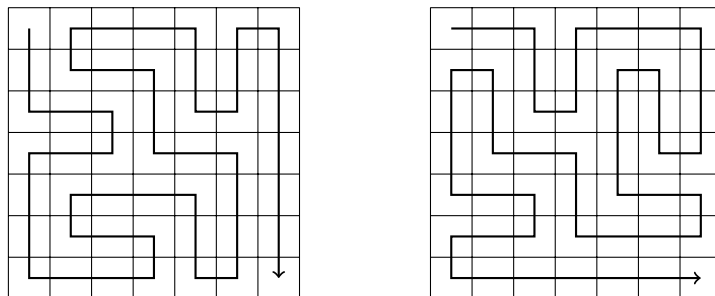
基本的なアルゴリズム - Basic algorithm

最初のア​​ルゴリズムは最適化をせずに単にバックトラックを使って、左上隅から右下隅への可能な経路をすべて生成し、そのような経路の数をカウントします。

- running time: 483 秒
- number of recursive calls: 760 億回

最適化 1 - Optimization 1

どの解法でも、一番最初は必ず 1 段下か右に移動します。この一方を考えると、グリッドの対角線に関して対称なもう一つのパスが必ず存在します。例えば、次のような経路です。

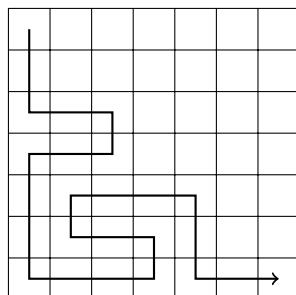


つまり、まず必ず 1 段階下 (または右) に移動したあとの解の数から 2 倍するとできるので計算すればいい量が半分になります。

- running time: 244 秒
- number of recursive calls: 380 億回

最適化 2 - Optimization 2

もし、他のマスすべてを訪問する前に右下のマ​​スに到達した場合、完成させることができないことは明らかです。

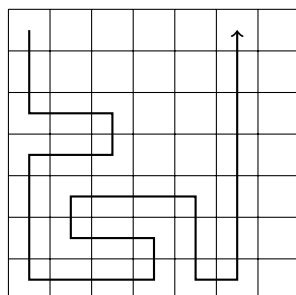


全マスの探索前に右下に到達した探索は打ち切ることにします。

- running time: 119 秒
- number of recursive calls: 200 億回

最適化 3 - Optimization 3

パスが壁に接触して左右に曲がることのできるような場合、グリッドは未訪問のマスを含む 2 つの部分に分割されます。次のような場合、道は左にも右にも曲がることができます。



ところが、どちらを選んでもすべてのマスを訪問することはできないので探索を打ち切ります。この最適化は非常に有効です。

- running time: 1.8 秒
- number of recursive calls: 2 億 2100 万回

最適化 4 - Optimization 4

最適化 3 を一般化します。もし、パスが壁以外で前に進むことができず左か右に曲がることのできる場合、やはりグリッドは 2 つに分かれてしまうので未訪問のマスが出来てしまいます。

次のような経路がこれにあたります。

なります。しかし、半分全列挙を用いると、より効率な $O(2^{n/2})$ で求めることができます。^{*2} ここで $O(2^n)$ と $O(2^{n/2})$ の時間計算量は全く異なることに注意してください。 $2^{n/2}$ は $\sqrt{2^n}$ に等しいことから明らかに計算量が違うことが分かるでしょう。

まず、リストを2つのリスト A 、 B に分割し、両方のリストに約半分の数字が含まれるようにします。最初の検索では、 A のすべての部分集合を生成し、その和をリスト S_A に格納します。同じようにリスト B からリスト S_B を作成します。このあと、 S_A と S_B からひとつずつ選びそれらの和が x になるようにできるかどうかを調べればよいです。これは、元のリストの数を用いて和 x を形成する方法がある場合に、まさに有効です。

例えば、リストが $[2, 4, 5, 9]$ で $x = 15$ だとします。まず、リストを $A = [2, 4]$ と $B = [5, 9]$ に分割しましょう。この後、リスト $S_A = [0, 2, 4, 6]$ と $S_B = [0, 5, 9, 14]$ を作成します。この場合、 S_A には和 6 が、 S_B には和 9 が含まれ、 $6 + 9 = 15$ となるので、 $x = 15$ を形成することが可能である。

このアルゴリズムは時間計算量が $O(2^{n/2})$ となるように実装することができます。まず、ソートされたリスト S_A と S_B を生成のはマージのような技法を用いて $O(2^{n/2})$ 時間で可能です。この後、リストはソートされているので、 S_A と S_B から和 x が生成できるかどうかを $O(2^{n/2})$ 時間で確認することができます。

^{*2} This idea was introduced in 1974 by E. Horowitz and S. Sahni [39].

第 6 章

貪欲アルゴリズム - Greedy algorithms

貪欲アルゴリズム **-greedy algorithm** は、常にその時点で最善と思われる選択をすることで、問題の解を構築する方法の総称です。貪欲なアルゴリズムは選択を取り消すことはなく、最終的な解を次々に直接構築していきます。このため、貪欲アルゴリズムは非常に効率的に動作します。

貪欲アルゴリズムの設計で難しいのは、常に問題の最適解を生成すると保証できるアルゴリズムを見つけ出すことです。貪欲アルゴリズムにおける局所的に最適な選択は大域的にも最適でなければならないからです。貪欲なアルゴリズムを証明するのはしばしば困難でもあります。

6.1 コイン問題 - Coin problem

コインの組が与えられ金額 n を形成する問題を考えましょう。コインの値は $\text{coins} = \{c_1, c_2, \dots, c_k\}$ であり、各コインは何度でも使用することができます。必要なコインの最小枚数は何枚か？

例えば、コインがユーロコイン（単位はセント）で、

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

のコインがあるとします。 $n = 520$ だとします。最適解は $200 + 200 + 100 + 20$ と少なくとも 4 枚のコインが必要になります。

貪欲なアルゴリズム - Greedy algorithm

この問題に対する貪欲アルゴリズムでは必要な金額が構成されるまで、常に可能な限り大きなコインを選択していきます。まず 200 セント硬貨を 2 枚選び、次に 100 セント硬貨を 1 枚、最後に 20 セント硬貨を 1 枚選ぶので、このアルゴリズムはうまく動作するように見えます。しかし、このアルゴリズムは常に正しく動作するのでしょうか？

コインがユーロコインであれば、貪欲アルゴリズムが常に機能すること、すなわち、コインの枚数が最も少ない解を常に生成することが判明しています。このアルゴリズムの正しさは、以下のように示されます。

各コイン 1, 5, 10, 50, 100 が最適解に現れるのはせいぜい 1 回で、もし解にそのようなコインが 2 枚含まれていれば次のように 1 枚ずつ減らし、よりよい解を得ることができるからです。例えば、解答にコイン $5+5$ が含まれている場合、10 に置き換えることができます。

同様に、2 と 20 が最適解に現れるのは最大 2 回で、コイン $2+2+2$ をコイン $5+1$ 、コイン $20+20+20$ をコイン $50+10$ と置き換えられます。また、 $2+2+1$ や、 $20+20+10$ の場合、5 や 50 で置き換えられます。

これらのことから、各コイン x について、 x より小さいコインだけを用いて最適に和 x またはそれ以上の和を構成することは不可能であることを示すことができます。例えば、 $x = 100$ の場合、小さいコインを使った最大の最適和は $50+20+20+5+2+2=99$ である。したがって、常に最大のコインを選択する貪欲なアルゴリズムが最適解を生成します。このようにアルゴリズム自体が単純であっても、貪欲なアルゴリズムが有効であることを主張するのは難しいことがわかる。

貪欲法でのコイン問題の一般化 - General case

一般的な場合、コインセットには任意のコインを入れることができ、貪欲アルゴリズムは、必ずしも最適解が得られるとは限りません。

貪欲なアルゴリズムが働かないことは、アルゴリズムが間違った答えを出す反例を示すことで証明できます。単純な例ではコインを $\{1, 3, 4\}$ とし、目標を 6 としましょう。最適解は $3+3$ であるのに貪欲アルゴリズムは解 $4+1+1$ を生成してしまいます。

一般的なコイン問題がどのような貪欲アルゴリズムで解けるかは不明です。^{*1}。しかし、第 7 章で見るように、常に正しい答えを与える動的計画法を使うと一般的な問題も効率的に解くことができます。

^{*1} However, it is possible to *check* in polynomial if the greedy algorithm presented in this chapter works for a given set of coins [53].

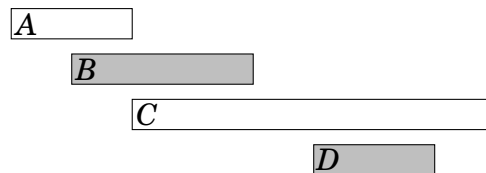
6.2 スケジューリング問題 - Scheduling

多くのスケジューリング問題も、貪欲アルゴリズムで解くことができ、古典的な問題としては以下のようなものです。 n 個のイベントとその開始時刻と終了時刻が与えられたとき、できるだけ多くのイベントを含むスケジュールを求めよ。ただし、イベントを部分的に選択することはできない。

例えば、次のようなイベントを考えられます。

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

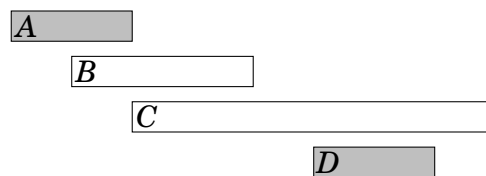
この場合は最大 2 つで *B* と *D* を次のように選択すればよいです。



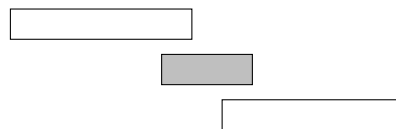
どのようなアルゴリズムが考えられ、どのようなものが最適なのでしょうか？

方法 1: Algorithm 1

最初に思い浮かぶのは最も短いイベントを選ぶことです。



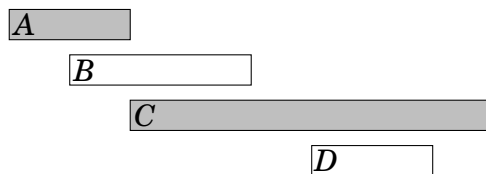
ところがこれは失敗するケースがあります。反例を示します。



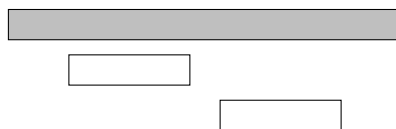
短いイベントを選ぶと 1 つしか選択できませんが、実際には長い 2 つを取ることが出来ることは明らかです。

方法 2: Algorithm 2

では、なるべく早い時間に始まるイベントを選び続けるのはどうでしょう？



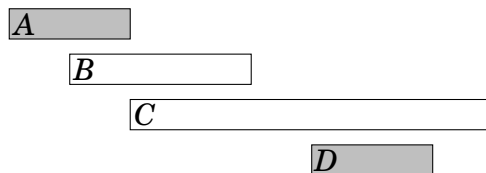
これも反例が簡単に見つかってしまいます。



最初のイベントを選択すると、他のイベントを選択することはできません。しかし、他の2つのイベントを選択することは可能です。

Algorithm 3

そこで3つ目のアイデアとしては、最も早く終わるイベントを選ぶアイデアはどうでしょうか？



このアルゴリズムは常に最適解を生成することができます。最初にできるだけ早く終了するイベントを選択することが常に最適な選択です。その後、同じ戦略で次のイベントを選択しそれ以上のイベントを選択できなくなるまで続けることが最適な選択となります。

このアルゴリズムを証明する方法を示します。最初にできるだけ早く終わる事象より、遅く終わる事象を選択した場合にどうなるかを考えてみましょう。次のイベントの選択の仕方は(最大でも)同じ数のイベントの選択肢しかありません。このため、遅く終わるイベントを選択してもより良い解が得られることはないので貪欲なアルゴリズムが正しいということになります。

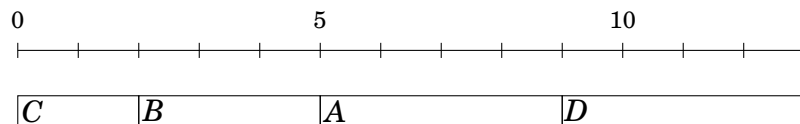
6.3 締め切り付きタスク問題 - Tasks and deadlines

期間と期限を持つ n 個のタスクが与えられ、タスクの実行順序を選択する問題を考えてみます。ここで各タスクについて $d-x$ 点のスコアを獲得するとします。 d はタスクの期限 x であるタスクが終了した時点とします。

例えば、以下のようなタスクがあるとする。

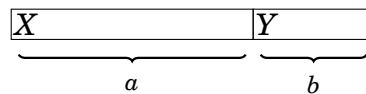
タスク	期間	期限
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

最適解は以下の通りです。

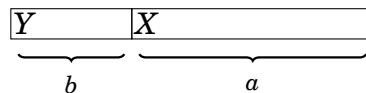


この解答では、*C* が 5 点、*B* が 0 点、*A* が -7 点、*D* が -8 点で合計で -10 点となります。

意外なことに、この問題の最適解は期限に全く依存せず、単にタスクにかかる時間順に並べたタスクを実行することが正しい貪欲な戦略となります。その理由は、2 つのタスクを次々に実行して最初のタスクが 2 番目のタスクより時間がかかるようなことがあれば、タスクを入れ替えた方が良くなるからです。例えば、次のようなスケジュールを考えてみます。



ここでは、 $a > b$ なので、タスクを入れ替える必要があります



今、*X* は b を獲得し、*Y* は a を獲得します。このため、 $a - b > 0$ となります。最適解では、連続する 2 つのタスクについて、短いタスクが長いタスクの前に来ることが成立しなければいけません。従って、タスクはその時間順に並べられなければならない。

6.4 最小和問題 - Minimizing sums

ここで、 n 個の整数 a_1, a_2, \dots, a_n が与えられたときに、次の和が最小になる x を求めたいとします。

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

ここでは、 $c = 1$ と $c = 2$ を例にとります。

Case $c = 1$

この場合は次の通りです。

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

[1, 2, 9, 2, 6] を入力すると、最適な解は $x = 2$ の時で、

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

となります。

一般的な場合、 x は数字の *median* - 中央値、つまりソート後の真ん中の数字が最適となります。例えば、[1, 2, 9, 2, 6] というリストは、並べ替えの結果 [1, 2, 2, 6, 9] なので、中央値である 2 を選びます。

x が中央値より小さければ、 x を大きくすれば合計は小さくなり、 x が中央値より大きければ、 x を小さくすれば合計は小さくなるから、最適解は x が中央値となります。 n が偶数で中央値が 2 つある場合は両方の値および、その間のすべての値が最適な選択となります。

Case $c = 2$

この場合は次の最小化です。

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

例えば、数字が [1, 2, 9, 2, 6] の場合、最適な解は $x = 4$ を選択することで、この場合、和は次のようになります。

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

一般的な場合に、 x は数字の平均を選ぶのが最適です。この例では、 $(1 + 2 + 9 + 2 + 6)/5 = 4$ です。この結果は、次のように和を提示することで導き出すことができる。

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

最後の部分は x に依存しないので、ここでは無視してもよいです。残りの部分は $s = a_1 + a_2 + \dots + a_n$ に対して、関数 $nx^2 - 2xs$ を形成します。これは $x = 0$ と $x = 2s/n$ の根を持つ上に開く放物線 (訳註: 二次関数) になるため、最小値は $x = s/n$ で、これは a_1, a_2, \dots, a_n の平均となります。

6.5 データ圧縮 - Data compression

バイナリコード - binary code とは、文字列の各文字に、ビットで構成される **コードワード - codeword** を割り当てたものです。各文字を対応するコードワードに置き換えることで、バイナリコードを使って文字列を圧縮することができます。例えば、次のバイナリコードでは、文字 A-D にコードワードを割り当てています。

character	codeword
A	00
B	01
C	10
D	11

これは各コードワードの長さが同じであることを意味する **定長 - constant-length** 符号と呼ばれます。例えば、AABACDACA という文字列を圧縮する場合、次のようになります。

000001001011001000

この符号を用いると、圧縮される文字列の長さは 18 ビットです。コードワードの長さが異なる **可変長 - variable-length** 符号を用いると、文字列をより圧縮することができます。よく登場する文字には短いコードワードを与え、めったに登場しない文字には長いコードワードを与えていきます。上記の文字列に対する最適な符号は、次のようになります。

character	codeword
A	0
B	110
C	10
D	111

最適な符号は、可能な限り短い圧縮文字列を生成でき、最適な符号を用いた圧縮文字列は

001100101110100,

と、18 ビットではなく 15 ビットになり 3 ビットを節約することができました。

コードワードは他のコードワードの接頭辞でないように決めることが重要です。例えば、あるコードが 10 と 1011 の両方のコードワードを含むことは許されません。この理由は、圧縮された文字列から元の文字列を生成できるようにしたいからです。もしコードワードが他のコードワードの接頭辞になり得るとしたら、これは常に可能とは限りません。

たとえば、次のようなコードワードにははいけません。

character	codeword
A	10
B	11
C	1011
D	111

圧縮された文字列 1011 が文字列 AB に対応するのか、文字列 C に対応するのが区別できないからです。

ハフマン符号化 - Huffman coding

ハフマン符号化 - Huffman coding^{*2} は、与えられた文字列を圧縮するために最適な符号を構築する貪欲なアルゴリズムです。このアルゴリズムは、文字列中の文字の頻度をもとに二分木を構築し、根から対応するノードへのパスを辿り各文字のコードワードを読み取ります。左への移動はビット 0 に対応し、右への移動はビット 1 に対応します。

初期状態では、文字列の各文字は、その文字が文字列中出现する回数を重みとするノードで表現します。次に、各ステップで重みが最小の 2 つのノードが結合され、元のノードの重みの合計を重みとする新しいノードを作成します。この処理をすべてのノードが結合されるまで続けます。

実際に AABACDACA という文字列に対して、ハフマン符号化がどのように最適な符号を生成するかを見ていきましょう。最初に、文字列の文字に対応する 4 つのノードが存在します。

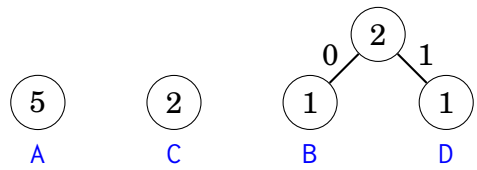


文字 A は文字列中に 5 回出現するので、文字 A を表すノードの重みは 5 です。他

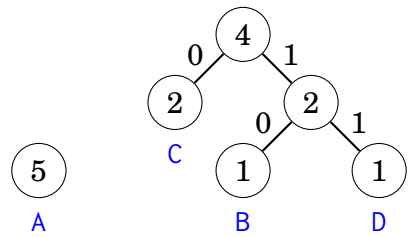
^{*2} D. A. Huffman discovered this method when solving a university course assignment and published the algorithm in 1952 [40].

の重みも同じように計算されます。

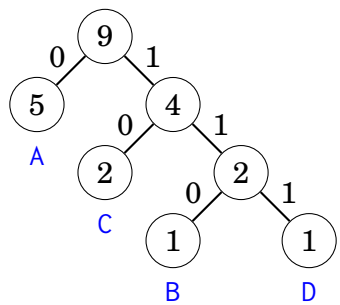
まず、重さ 1 の文字 B と文字 D に対応するノードを結合した結果がこれです。



重み 2 のノードを結合します。



最後に残った 2 つのノードを結合します。



すべてのノードがツリー上に揃ったので、コードワードをツリーから読み取りましょう。

character	codeword
A	0
B	110
C	10
D	111

第 7 章

動的計画法 - Dynamic programming

動的計画法 - Dynamic programming とは全探索の正しさと貪欲アルゴリズムの効率性を併せ持つ手法です。動的計画法が適応できる場面とは、問題を独立に解くことのできる重複する部分問題に分割できる場面です。

動的計画法には大きく 2 つの使い方がある。

- **最適解を見つける:** できるだけ大きな解あるいは小さな解を見つけたい。
- **解の数を数える:** 可能な解の総数を計算したい。

動的計画法の理解は、競技プログラマのキャリアのマイルストーンであるともいえます。基本的な考え方は簡単ですが、様々な問題に対して動的計画法をどのように適用するかが課題となります。この章では、出発点となる古典的な問題を紹介します。

7.1 コイン問題 - Coin problem

第 6 章ですでに見た問題ですが、再度注目しましょう。この問題はコインの値の集合 $\text{coins} = \{c_1, c_2, \dots, c_k\}$ と目標金額 n が与えられたとき、できるだけ少ないコインで合計 n を形成することが課題でした。

第 6 章では、常に可能な限り大きなコインを選ぶ貪欲なアルゴリズムを使って問題を解きました。貪欲アルゴリズムは、コインがユーロコインの場合などは有効ですが、一般的な場合では必ずしも最適解が得られるとは限らないということも説明しました。

動的計画法を用いてどんなコインセットにも対応できるように効率的に問題を

解決していきます。動的計画法では総和を求めるために全探索のアルゴリズムと同様、すべての可能性を検討する再帰的な関数を用います。この時にメモ化を使用して各部分問題の答えを一度だけ計算して効率的に動作させます。

再帰的な式 - Recursive formulation

動的計画法の考え方は、問題を再帰的な式として表現し、より小さな部分問題の解から問題の解を計算できるようにしていきます。コイン問題の場合、次のような再帰的な問題とします。「和 x を形成するのに必要な最小のコインはいくつですか？」

$\text{solve}(x)$ を和 x に必要なコインの最小枚数とすると、関数の値はコインの値に依存します。例えば、 $\text{coins} = \{1, 3, 4\}$ のとき、関数の最初の値は以下の通りです。

$\text{solve}(0)$	$=$	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、 $\text{solve}(10)=3$ は、和が 10 になるためには、少なくとも 3 枚のコインが必要ということを意味します。最適解は $3+3+4=10$ です。

solve の本質的な特性は、その値を再帰的に計算できることで、小さいコインの和から注目していきます。例えば、上記のシナリオでは、最初のコインは 1、3、4 です。最初にコイン 1 を選んだ場合、残っている課題は最小数のコインを用いて 9 を形成することとなります。これは元の問題の下位問題となります。コイン 3、4 についても同様です。したがって、最小のコインの枚数を計算するには、次のような再帰的な公式を用いることができます。

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

再帰の特殊な定式は $\text{solve}(0)=0$ であり、これは空の和を形成するためにコインは必要ないからです。

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

これで、 x のための最小枚数を計算する一般的な再帰関数が準備できました。

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

$x < 0$ の場合、負の金額を形成することは不可能であるため値は ∞ としましょう。次に、 $x = 0$ のとき、値は 0 です。なぜなら、負の金額を形成するためにコインは必要ためです。最後に $x > 0$ の場合、変数 c は最初のコインをどのように選ぶかについてすべての可能性を調べます。

問題を解く再帰の式が見つかれば、C++ で直接解を実装することができる (定数 INF は無限大です)。

```
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;
}
```

この関数は効率的とは言えません。なぜなら、和を構成する方法は指数関数的に増えるためです。そこでメモ化と呼ばれる技術を使って、この関数を効率的にする方法を探ります。

メモ化の利用 - Using memoization

動的計画法の重要な考え方は**メモ化 - memoization**を用いて再帰的な関数の値を効率的に計算することです。つまり、一度計算した値はその関数用の構造に記憶します。各パラメータに対して、関数の値は再帰的に計算して記録します。次からは配列から直接値を参照します。

この例では配列を使用します

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

`ready[x]`は`solve(x)`の値が計算されたかどうかを示し、計算された`value[x]`にその値が格納される。定数 N は、必要な値がすべて配列に収まるように選ばれています。

この関数は次のように効率的に実装できるようになりました。

```
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;
}
```

この関数は、前と同様に $x < 0$ と $x = 0$ の基本ケースを処理します。次に、`ready[x]`を使って`value[x]`に`solve(x)`が既に格納されているかどうかをチェックして、格納されていればそれを直接返します。そうでない場合は、`solve(x)`の値を再帰的に計算し`value[x]`に格納します。

この関数は、各パラメータ x に対する答えを一度だけ再帰的に計算するため、効率的に動作します。最初に`solve(x)`の値が`value[x]`に格納されるため、 x を指定して効率的に取り出すことができます。このアルゴリズムの時間計算量は n を目標和 k をコインの枚数とすると、 $O(nk)$ です。

なお、パラメータ $0 \dots n$ に対して`solve`の全値を単純に計算するループを使って、配列の値を反復的に構築しても良いです。

```
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);
        }
    }
}
```

多くの競技プログラマは、この実装の方が短く定数倍の計算量も低いので、好ん

で使っています。これ以降、私たちの例でも反復処理の実装を使用します。しかし、動的計画法の解法は再帰的な関数で考える方が簡単な場合が多いというのは覚えておいてください。

構築 - Constructing a solution

しばしば、最適解の値だけではなく、どのように構成されるかの例を示すことの両方を求められることがあります。これに応えるため、コインの問題では各金額について、最適解の最初のコインを示す別の配列を宣言することができます。

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

アルゴリズムを次のように改変しましょう。

```
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;
        }
    }
}
```

以下のように和 n の最適解に現れるコインを表示させることができます。

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

解のカウント - Counting the number of solutions

ここで、コイン問題の別バージョンとして、コインを用いて和 x を生み出す方法の総数を計算する問題を考えます。たとえば、 $\text{coins} = \{1, 3, 4\}$, $x = 5$ とすると合計 6 通りあることになる。

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

ここでも、再帰的に問題を解くことができます。例えば、`coins = {1,3,4}` の場合、`solve(5) = 6` となり、再帰的な式は次のようになる。

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

一般的な再帰関数は次のように示せます。

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

$x < 0$ の場合、解が存在しないので値は 0 である。 $x = 0$ の場合、空の和を形成する方法は 1 つしかないから値は 1 です。それ以外の場合は c をコインとして、`solve(x-c)` の形のすべての値の合計を計算する。

次のコードは、 $0 \leq x \leq n$ である時の `solve(x)` の値 `count` を構築します。

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

また、解の数が非常に多く、正確な数を計算する必要はないが、例えば $m = 10^9 + 7$ のように、 m のモジュロで答えを出せば十分なことがよくあります。これは、すべての計算がモジュロ m で行われるようにコードを変更することで求められます。

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

の後に次のようにします。

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

さて、ここまでで動的計画法の基本的な考え方はすべて説明しました。動的計画法はいろいろな場面で使えるので、これから動的計画法の可能性について、さらに例を示す問題を見ていきましょう。


7.2 最長増加部分列 (LIS) - Longest increasing subsequence

最長増加部分列 (LIS) - longest increasing subsequence とは n 個の要素を持つ配列の中で、最も長く増加する部分列を求める問題です。言い換えると左から右へ向かう配列要素の最大長の並びで、並びの各要素は前の要素より大きくなるようなものです。

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

この配列は最も長い増加する部分列が 4 個の要素を含むことを意味します。

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



$\text{length}(k)$ を k 文字目まで見た時の LIS としましょう。したがって $0 \leq k \leq n-1$ となる $\text{length}(k)$ の値をすべて計算すれば、最も長く増加する部分列の長さが求められます。例えば、上の配列に対する関数の値は以下の通りである。

```
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、 $\text{length}(6) = 4$ は位置 6 で終わる最も長い増加部分配列が 4 要素で構成されることを示します。

$\text{length}(k)$ の値を計算するには、 $\text{array}[i] < \text{array}[k]$ であって、 $\text{length}(i)$ ができるだけ大きくなる位置 $i < k$ を見つければよいです。そうすると、 $\text{length}(k) = \text{length}(i) + 1$ となり、これが $\text{array}[k]$ を部分配列に追加する最適な方法であるとわかります。しかし、そのような位置 i がない場合には $\text{length}(k) = 1$ となり部分配列には $\text{array}[k]$ しか含まれないことになります。

関数のすべての値はその小さい値から計算できるので、動的計画法を使うことができます。次のコードでは、関数の値は配列 `length` に格納されます。

```
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2つのネストされたループで構成されているため $O(n^2)$ で動作します。しかし、動的計画法の計算をより効率的に $O(n \log n)$ 時間で実装することも可能です。あなたはこの方法を見つけることができますか？

7.3 グリッド上のパス - Paths in a grid

次の問題は、 $n \times n$ のグリッドの左上隅から右下隅まで、下と右にしか移動しないような経路を見つける問題です。各マスには正の整数が含まれており、経路に沿った値の合計ができるだけ大きくなるように経路を構成したいとします。

下図は、格子状のマス例とその最適経路を示したものです。

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

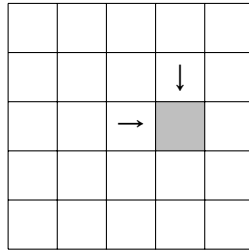
パス上の値の合計は 67 であり、これは左上隅から右下隅までのパスで可能な最大の合計である。

格子の行と列には 1 から n までの番号が振られ、 $value[y][x]$ はマス (y, x) の値に等しいとしましょう。 $sum(y, x)$ は左上隅から (y, x) までの経路上の最大和を表します。ここで、 $sum(n, n)$ は左上隅から右下隅までの最大和を表します。例えば、上の格子では、 $sum(5, 5) = 67$ であると言えます。

この問題は以下のように再帰的に総和を計算することができます。

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

何故なら、マス (y, x) への道は次のように $(y-1, x)$ か $(y, x-1)$ からくるという観察に基づきます。



つまり、この式は和を最大化する方向を選択します。 $y=0$ か $x=0$ なら その方向からの入力 $\text{sum}(y,x)=0$ として計算します。

関数 sum は 2 つのパラメータを持つので動的計画法配列も 2 次元になります。

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

を次のように計算していきます。

```
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];
    }
}
```

これは $O(n^2)$ で計算可能です。

7.4 ナップザック問題 - Knapsack problems

knapsack - ナップザック問題 とは、集合が与えられ、ある性質を持った部分集合を見つけなければならない問題のことです。ナップザック問題は多くの場合動的計画法を用いて解くことができます。

重みのリスト $[w_1, w_2, \dots, w_n]$ が与えられたとき、その重みを使って構成できるすべての和を決定することなどが例に挙げられます。例えば、重みが $[1, 3, 3, 5]$ の場合、以下の和が可能です。

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

この場合、2 と 10 を除く $0 \dots 12$ の間のすべての和が実現可能であることを意味します。例えば、和 7 は、重み $[1, 3, 3]$ を選んで実現可能です。

この問題を解くために、最初の k 個の重みだけを用いて和を構成する部分問題に注目しましょう。最初の k 個の重みを使って和 x を構成できる場合は $\text{possible}(x, k) = \text{true}$ 、そうでない場合は $\text{possible}(x, k) = \text{false}$ とします。この関数の

値は以下のように再帰的に計算することができます。

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

この式は、和に重み w_k を使うか使わないかのどちらかを選ぶことができることに基づいています。 w_k を使う場合、残りのタスクは最初の $k - 1$ 個の重みを使って和 $x - w_k$ を形成することであり、 w_k を使わない場合、残りのタスクは最初の $k - 1$ 個の重みを使って和 x を形成することになります。

また、初期状態として

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

が与えられます。というのは、重みがない場合は、和は 0 にしかならないからです。

次の表は、重み [1, 3, 3, 5] に対する関数のすべての値を示しています (記号 "X" は真の値を示す)。

$k \backslash x$	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た後、 $\text{possible}(x, n)$ は、すべての重みを使って和 x を構成できるかどうかを教えてくれる。

W は重みの総和を表すとする。以下の $O(nW)$ 時間の動的計画法が再帰関数に相当します。

```
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];
    }
}
```

さて、ここでは、和が x の部分集合を構成できるかどうかを示す 1 次元配列 $\text{possible}[x]$ のみを用いるよりよい実装を示しましょう。この配列は、新しい重みごとに右から左へと更新します。

```
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;
    }
}
```

なお、ここで紹介した一般的な考え方は、多くのナップザック問題で利用することができます例えば、重みと価値を持つオブジェクトが与えられた場合に、各重み和について部分集合の最大の価値を決定することができます。

7.5 編集距離 - Edit distance

編集距離 - edit distance あるいは **レーベンシュタイン距離 - Levenshtein distance**^{*1} とは、ある文字列を別の文字列に変換するために必要な編集操作の最小回数のことである。編集操作とは以下の通りである。

- 挿入 (e.g. ABC → ABCA)
- 削除 (e.g. ABC → AC)
- 変更 (e.g. ABC → ADC)

例えば、LOVE と MOVIE の編集距離は 2 です。これは、まず LOVE → MOVE(変更) の操作を行い、次に MOVE → MOVIE(挿入) の操作で一致させられます。1 つの操作だけでは不十分であることが明らかであるため可能な限り少ない操作回数であることは明らかでしょう。

長さ n の文字列 x と長さ m の文字列 y が与えられたときの x と y の編集距離を計算します。この問題を解決するために、接頭辞 $x[0\dots a]$ と $y[0\dots b]$ の編集距離を与える関数 $\text{distance}(a, b)$ を定義します。この関数を用いると、 x と y の編集距離は $\text{distance}(n-1, m-1)$ に等しくなる。

distance は次のように計算することができます。

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

ここで、 $x[a] = y[b]$ ならば $\text{cost}(a, b) = 0$ 、そうでなければ $\text{cost}(a, b) = 1$ とします。この式では、文字列 x の編集方法として、次のようなものを考えている。

^{*1} The distance is named after V. I. Levenshtein who studied it in connection with binary codes [49].

- $\text{distance}(a, b-1)$: x の最後に文字の挿入を行う
- $\text{distance}(a-1, b)$: x の文字を最後を削除する
- $\text{distance}(a-1, b-1)$: x の最後の文字が一致あるいは変更する

最初の2つのケースでは、1つの編集操作(挿入または削除)が必要です。最後の場合、 $x[a] = y[b]$ であれば、編集なしで最後の文字をマッチングさせることができ、そうでなければ、1つの編集操作(modify)が必要になります。次の表は、例の場合の distance の値を示したものです。

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

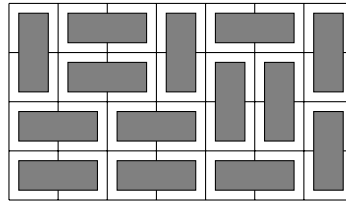
表の右下には、LOVE と MOVIE の編集距離が2であることが示されています。この表は最短の編集操作の順序を構築する方法も示されています。この場合の経路は次のようになります。

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

LOVE and MOVIE は最後の文字が等しいので、両者の編集距離は LOV と MOVI の編集距離と等しくなる。MOVI から I という文字を削除するには、1回の編集操作で可能である。したがって、編集距離は LOV と MOV の間の編集距離より1大きいといったように読み取ります。

7.6 タイルの数え上げ - Counting tilings

動的計画法の計算の過程は、単に固定された数の組み合わせよりも複雑な場合があります。例えば、 1×2 と 2×1 のタイルを使って、 $n \times m$ の格子を何通り埋めることができるかを計算する問題を考えてみよう。例えば、 4×7 グリッドの有効な解は以下の通りです。



この場合、総解答数は 781 となります。

この問題は動的計画法を用いて、グリッドの行を 1 つずつ調べていくことで解くことができます。解の各行は、集合 $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$ から m 個の文字を含む文字列として表すことができます。例えば、上の解答は 4 つの行からなり、それらは以下の文字列に対応する。

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

$\text{count}(k, x)$ は、文字列 x が k 行目に対応するようなグリッドの $1 \dots k$ 行目の解を構成する方法の数を表すとしましょう。ある行の状態は前の行の状態によってのみ制約されるので動的計画法を使用することができます。

1 行目に \sqcup という文字がなく、 n 行目に \sqcap という文字がなく、かつ連続するすべての行が対応を持っていれば、解は成立する。例えば $\sqcup \sqsubset \sqsubset \sqcup \sqcap \sqcap \sqcup$ と $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$ の行は対応があり、 $\sqcap \sqsubset \sqsubset \sqcap \sqsubset \sqsubset \sqcap$ と $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$ の行は対応がない。

1 行は m 個の文字からなり、各文字には 4 つの選択肢があるので、異なる行の数は最大でも 4^m です。したがって、各行について $O(4^m)$ 個の可能な状態を調べ、それぞれの状態について、前の行について $O(4^m)$ 個の可能な状態があるので、解法の時間計算量は $O(n4^{2m})$ となります。係数 4^{2m} が時間計算量を支配するので、短い方の辺の長さが m になるようにグリッドを回転させるのがよいでしょう。

行をよりコンパクトに表現することで、より効率的に解を得ることができます。前の行のどの列が縦長のタイルの上側の正方形を含むかが分かればよいと分かったので、 \sqcap と \sqcup だけで表現ができます。 \sqsubset は、 $\sqcup, \sqsubset, \sqsubset$ の組み合わせである。この表現を用いると、 2^m 個の行を表現できるため、 $O(n2^{2m})$ で良いです。

最後に、タイリングを求めるためには驚くべき公式があります。^{*2}

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

^{*2} Surprisingly, this formula was discovered in 1961 by two research teams [43, 67] that worked independently.

この式は、タイリングの数を $O(nm)$ で計算するため、非常に効率的です。ただし、答えが実数の積であるため、この式を使う場合は中間結果をいかに正確に保存するかが問題となります。

第 8 章

ならし解析 - Amortized analysis

多くのプログラムにおいて、時間計算量は、アルゴリズムの構造を調べるだけで簡単に解析できます。例えば何回ループが実行されるかといった解析を行います。ですが、直感的な解析だけでは計算量を正しく見積もれないこともあります。

ならし解析 - Amortized analysis を使って、時間計算量が一定に定まらないような計算量を見積もることができます。基本的なアイデアは、個々の操作に注目するのではなく、アルゴリズムの実行中に全ての操作に使用される操作の合計時間を推定することです。

8.1 2 ポインタ - Two pointers method

2 ポインタテクニック - two pointers method では、決められた方向にしか動かない 2 つのポインタを用いて処理を行います。これにより、アルゴリズムの最大の動作時間が保証されます。この例を 2 つ見ていきましょう。

連続部分配列の和 - Subarray sum

n 個の正の整数からなる配列と x が与えられたとき、和が x となる連続した部分配列を見つけるか、そのような部分配列はないと出力する問題を考えてみます。

例えば配列は以下のように与えられます。

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

例えば以下のように和が 8 となる連続部分配列があります。

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

この問題は 2 ポインタを使うことで $O(n)$ 時間で解くことができます。これは、

部分配列の最初と最後の場所を指すポインタを保持し、各ステップで右ポインタを右にシフトしても和が x 以下であるなら、ポインタを右にシフトします。そうでなければ左ポインタを左にシフトします。和がちょうど x になれば解が見つかったことになります。

ここで、 $x=8$ を達成するための操作を見てみます。

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

まず、いくつか右ポインタを進め、1, 3, 2 まで進めると和は 6 です。

1	3	2	5	1	1	2	3
↑		↑					

さらに右のポインタを進めると x を超えてしまうので、左のポインタを右に動かすことにします。

1	3	2	5	1	1	2	3
	↑	↑					

この状態でも 5 を加えると x を超えてしまうので、左のポインタをもう 1 つ右に進めます。これで右のポインタを右に進められることになり $2+5+1=8$ となって x を見つけることができました。

1	3	2	5	1	1	2	3
		↑	↑				

このアルゴリズムの実行時間の見積もりには右ポインタの移動ステップ数に注目しましょう。ポインタが 1 回のターンで何ステップ動けるかというのは配列の値によって異なります。ただしポインタは右にしか動かないので、アルゴリズム中に合計 $O(n)$ ステップしか動けないことに注目しましょう。同様に左のポインタはアルゴリズム中に $O(n)$ ステップ移動するのでアルゴリズム全体は $O(n)$ で動作します。

2SUM problem

別の問題を考えましょう。 n 要素の配列と x が与えられたとき、和が x となる配列の 2 つの要素を見つけるか、そのような値は存在しないことを出力しなさい、という問題です。これを効率的に解くには配列の値を昇順に並べ替えます。次に、2 つのポインタを用いて配列を繰り返し処理します。左のポインタは最初の値から始まり、1 ターンごとに 1 ステップ右へ移動させます。右のポインタは最後の値から始まり、左と右の値の合計が最大で x になるまで常に左に移動します。例えば、次の

ような配列で $x = 12$ であるケースを考えましょう。

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

ポインタの初期位置は以下の通りです。値の和は $1 + 10 = 11$ で x より小さいです。

1	4	5	6	7	9	9	10
↑							↑

そこで、左のポインタが右に 1 ステップ移動します。ここで、右のポインタは左に 3 ステップ移動し、合計は $4 + 7 = 11$ となる。

1	4	5	6	7	9	9	10
	↑			↑			

左ポインタは再び右へ 1 ステップ移動します。 $5 + 7 = 12$ という解が存在することが分かりました。

1	4	5	6	7	9	9	10
		↑		↑			

実行時間を見積もります。まず、配列を $O(n \log n)$ でソートし、次に両方のポインタを $O(n)$ 回移動させるため全体では $O(n \log n)$ となります。二分探索を用いれば、別の方法で $O(n \log n)$ 時間で解くこともできます。配列の各値に対して和 x となる値を二分探索すればよいです。

これより難しい問題としては、和が x となる 3 つの配列値を求める **3SUM 問題** があります。上記のアルゴリズムの考え方をを用いると、この問題はに従って $O(n^2)$ 時間で解くことができます。^{*1}

8.2 最も近い小さな要素 - Nearest smaller elements

データ構造に対して行われた操作の回数を推定するために、ならし解析がよく利用されます。演算の実行は入力によって不均等に分布するので、アルゴリズムの特定のフェーズで演算が発生することとなりますが演算の総数には限りがあります。

ここで配列の各要素について既出の最も近い小さい要素、つまり配列内でその要素に先行する最初の小さい要素を見つける問題を考えてみます。そのような要素が

^{*1} 長い間、3SUM 問題を $O(n^2)$ 時間より効率的に解くことは不可能と考えられていました。しかし、2014 年に、そうではないことが判明した [30]

存在しないこともあり得ますが、その場合はアルゴリズムがそのことを報告するとしましょう。まずは、この問題をスタック構造を用いてどのように効率的に解決できるかを見ていきます。

配列を左から右へ走査し、配列要素のスタックを維持します。配列の各位置で、一番上の要素が現在の要素より小さくなるか、スタックが空になるまで、スタックから要素を取り除いていきます。そして、一番上の要素が現在の要素に最も近い小さい要素であることを報告します。また、スタックが空であれば、そのような要素は存在しません。そして、現在の要素をスタックに追加するといった操作を続けます。

例として、次のような配列を考えてみましょう。

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

まず、各要素が前の要素より大きいので、1、3、4の要素がスタックに追加されます。したがって、4の最も近い小さい要素は3であり、3の最も近い小さい要素は1となります。

1	3	4	2	5	3	4	2
1	→	3	→	4			

次の要素である2は、スタックの一番上の2つの要素より小さい。このため、要素3と4がスタックから **pop** し次に要素2がスタックに追加されます。その最も近い小さい要素は1となります。

1	3	4	2	5	3	4	2
1	→		2				

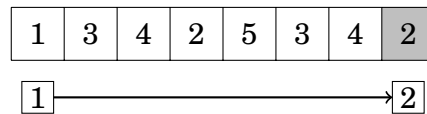
そして、要素5は要素2より大きいのでスタックに追加され、その最も近い小さい要素は2となります。

1	3	4	2	5	3	4	2
1	→		2	→	5		

この後、要素5をスタックから取り除き、要素3、4をスタックに追加しましょう。

1	3	4	2	5	3	4	2
1	→		2	→	3	→	4

最後に、1 以外のすべての要素がスタックから取り除かれ、最後の要素 2 がスタックに追加されます。



アルゴリズムは入力によるスタックオペレーションの総数に依存します。現在の要素がスタックの一番上の要素より大きい場合、直接スタックに追加され効率的です。ところが、スタックにいくつかの大きな要素が含まれることがあり、それらを取り除くのに時間がかかるように思えます。ですが、ここで注目すべきなのは各要素はスタックに最大一回追加され最大一回削除されるという点です。つまり、各要素は $O(1)$ スタック操作を引き起こすのでアルゴリズムは $O(n)$ 時間で動作するといえます。

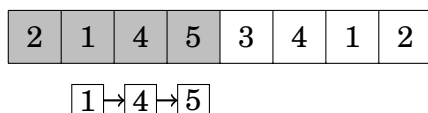
8.3 最小スライディングウィンドウ - Sliding window minimum

スライディングウィンドウ - sliding window は、配列の左から右へ移動する一定サイズの部分配列のことです。各ウィンドウの位置で、窓の中の要素について何らかの情報を計算します。ここでは、**最小スライディングウィンドウ - sliding window minimum** を維持する問題、つまり、各ウィンドウ内の最小値を報告することに焦点を当てます。

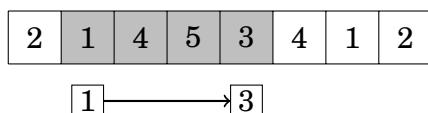
スライディングウィンドウの最小値は、前述の最も近い小さい要素を計算するのに用いたのと同様の考え方で計算することができます。各要素が前の要素より大きい待ち行列を維持し、最初の要素は常に窓の内側の最小要素に対応するようにします。各ウィンドウの移動後、最後のキューの要素が新しいウィンドウの要素より小さくなるか、キューが空になるまで、キューの末尾から要素を削除していきます。また、最初の待ち行列の要素がもうウィンドウの中にない場合は、それを削除します。最後に、新しいウィンドウの要素を待ち行列の最後に追加します。例として、次のような配列を考えてみましょう。

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

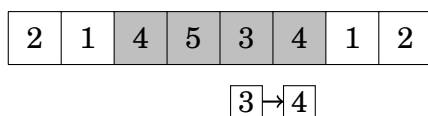
スライディングウィンドウの大きさを 4 とすると、最初のウィンドウの位置で最小の値は 1 です。



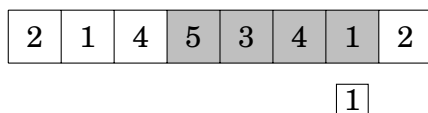
まず、ウィンドウは1ステップ右に移動します。新しい要素3は待ち行列の要素4と5より小さいので、要素4と5は待ち行列から取り除かれ、要素3が待ち行列に追加されます。最小の値はまだ1のままです。



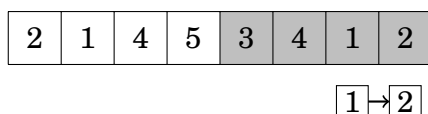
この後、再びウィンドウが移動し、最小の要素1はウィンドウに属さなくなります。従って、これはキューから取り除かれ、最小の値は3になります。そして、新しい要素4がキューに追加されます。



次の新しい要素1は、待ち行列のすべての要素より小さいです。このため、キューからすべての要素が削除され、要素1だけが含まれるようになります。



最後に、ウィンドウは最後の位置に到達します。要素2がキューに追加されるが、ウィンドウ内の最小値はまだ1のままです。



各配列要素は正確に一度だけキューに追加され、最大一度だけキューから削除されるので、このアルゴリズムは $O(n)$ 時間で動作します。

第 9 章

区間クエリ - Range queries

この章では区間クエリ - **range query** を効率的に処理できるデータ構造を取り上げます。区間クエリでは配列の部分配列に対してなんらかの値を計算することが課題となります。典型的なレンジクエリをいくつか紹介します。

- $\text{sum}_q(a, b)$: $[a, b]$ の和を求めます
- $\text{min}_q(a, b)$: $[a, b]$ に含まれる最小の値を求めます
- $\text{max}_q(a, b)$: $[a, b]$ に含まれる最大の値を求めます

以下の $[3, 6]$ の区間を考えます。

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

この例では $\text{sum}_q(3, 6) = 14$, $\text{min}_q(3, 6) = 1$, $\text{max}_q(3, 6) = 6$ となります。

区間クエリを処理する簡単な方法は区間の部分配列すべての配列値をループで計算することです。たとえば **sum** の区間クエリは以下のように書けます。

```
int sum(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++) {
        s += array[i];
    }
    return s;
}
```

この関数は配列のサイズ n に対して $O(n)$ 時間で動作します。つまり q 個のクエリがあれば $O(nq)$ となるため、 n と q が大きいと計算は遅くなります。この章ではより効率的なレンジクエリの求め方を見ていきましょう。

9.1 静的なクエリ - Static array queries

まず、配列が**静的** - *static* である場合を考えます。静的な配列とは、クエリで配列の値が更新されることがない場合を指します。どのようなクエリに対しても高速に答えを教えてくれる静的なデータ構造を構築すれば十分です。

和のクエリ Sum queries

静的配列に対する sum クエリは**累積和** - **prefix sum array** の配列を事前に作成しておく簡単にクエリできます。累積和の配列とは位置 k に対して $\text{sum}_q(0, k)$ とする配列です。これは $O(n)$ 時間で構築できます。

次のような配列があったとします。

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

これに対応する累積和の配列は以下の通りです。

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

累積和の配列には $\text{sum}_q(0, k)$ のすべての値が含まれているので、以下のように任意の区間の和 $\text{sum}_q(a, b)$ を $O(1)$ 時間で計算することが可能です。

$$\text{sum}_q(a, b) = \text{sum}_q(0, b) - \text{sum}_q(0, a - 1)$$

$\text{sum}_q(0, -1) = 0$ と定義しておけば $a = 0$ のときにも上式が成り立ちます。例を見ていきます。[3, 6] の区間を考えてみましょう。

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

この場合 $\text{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$ ですが、この和を以下のように累積和の配列の 2 つの値から計算できます。

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

$\text{sum}_q(3, 6) = \text{sum}_q(0, 6) - \text{sum}_q(0, 2) = 27 - 8 = 19$. と灰色の部分の値を使って求めることができました。

このアイデアは多次元に一般化することもできます。例えば、任意の矩形部分配列の和を $O(1)$ 時間で計算できる 2 次元累積和配列を構築します。事前計算として

左上からの累積された和をもつ 2 次元累積和配列を作っておきます。

次の灰色部分の値の和を求めたいとします。

		<i>D</i>				<i>C</i>			
		<i>B</i>				<i>A</i>			

これは、以下の式で計算できます。

$$S(A) - S(B) - S(C) + S(D),$$

ここで、 $S(X)$ は左上から X の位置までの長方形部分の和です。

最小クエリ - Minimum queries

最小クエリは和クエリよりも処理が難しいです。それでも $O(n \log n)$ の非常に簡単な前処理をしておくと $O(1)$ で任意の区間の最小クエリに答えることができます。^{*1} ここでは最小だけを説明しますが最大も同じように考えられます。

区間の最小値のクエリに対するアイデアは $b - a + 1$ (区間の長さ) が 2 の累乗である $\min_q(a, b)$ のすべての値を事前に計算しておくスパーステーブルと呼ばれるデータ構造です。

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

この場合、以下のようにテーブルを持ちます。

^{*1} This technique was introduced in [7] and sometimes called the **sparse table** method. There are also more sophisticated techniques [22] where the preprocessing time is only $O(n)$, but such algorithms are not needed in competitive programming.

a	b	$\min_q(a, b)$	a	b	$\min_q(a, b)$	a	b	$\min_q(a, b)$
0	0	1	0	1	1	0	3	1
1	1	3	1	2	3	1	4	3
2	2	4	2	3	4	2	5	1
3	3	8	3	4	6	3	6	1
4	4	6	4	5	1	4	7	1
5	5	1	5	6	1	0	7	1
6	6	4	6	7	2			
7	7	2						

2 の累乗である $O(n \log n)$ 個の区間に対応するテーブルが存在するため、事前計算される値の数は $O(n \log n)$ 個です。この事前計算のテーブルは再帰的な式を使って効率的に計算することができます。

$$\min_q(a, b) = \min(\min_q(a, a + w - 1), \min_q(a + w, b)),$$

ここで $b - a + 1$ を 2 の累乗として $w = (b - a + 1)/2$ とします。このテーブルの値をすべて計算するには $O(n \log n)$ の時間がかかります。

この後、 $\min_q(a, b)$ の任意の値は、事前に計算された 2 つの値の最小値として $O(1)$ で計算できます。 $b - a + 1$ を超えない最大の 2 の累乗を k としましょう。 $\min_q(a, b)$ の値は次の式で計算することができます。

$$\min_q(a, b) = \min(\min_q(a, a + k - 1), \min_q(b - k + 1, b)).$$

区間 $[a, b]$ というのは長さ k の区間 $[a, a + k - 1]$ と $[b - k + 1, b]$ のを結合したものです。

[1,6] の区間を考えてみましょう。

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

区間の長さは 6 なので 6 を超えない 2 の最大累乗は 4 です。区間 [1, 6] は区間 [1, 4] と [3, 6] を結合したものとして求められます。

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

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

$\min_q(1, 4) = 3$ で $\min_q(3, 6) = 1$ のため、両区間を含むクエリは $\min_q(1, 6) = 1$ と求め

られました。

9.2 BIT - Binary indexed tree

BIT - binary indexed tree あるいは **フェニック木 - Fenwick tree**^{*2} は動的な操作が可能な累積和の配列です。この構造は配列に対して区間和のクエリと単一値の更新を $O(\log n)$ で操作できます。

BIT の利点はこれは先ほど述べた累積和では不可能なクエリの途中で配列の値の更新が効率的に行えます。累積和では更新のたびに配列全体を再び構築する必要があり、 $O(n)$ 時間かかるからです。

構造 - Structure

BIT は *binary indexed tree* という (木のつく) 名前ではありますが配列で表現されます。また、実装を容易にするために BIT では配列が 1 インデックスで扱うことに注意してください。

$p(k)$ は k を割る 2 の最大累乗を表すとしましょう。この時に以下のようになります。

$$\text{tree}[k] = \text{sum}_q(k - p(k) + 1, k),$$

各位置 k には、元の配列の長さを $p(k)$ とし、位置 k で終わる区間までの値の合計が含まれます。例えば、 $p(6) = 2$ なので、 $\text{tree}[6]$ には $\text{sum}_q(5, 6)$ の値が含まれます。

例えば、次のような配列を考えてみましょう。

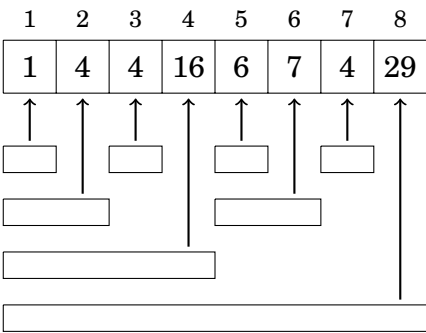
1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

これに対応する BIT は次のようになります。

1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

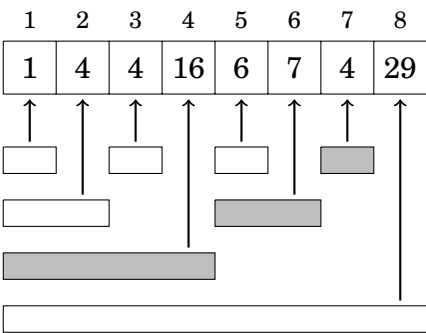
元の図との対応をわかりやすく表現したのが次の図です。

^{*2} The binary indexed tree structure was presented by P. M. Fenwick in 1994 [21].



BIT を用いると $\text{sum}_q(1, k)$ の任意の区間の値を $O(\log n)$ 時間で計算できます。区間 $[1, k]$ が常に $O(\log n)$ の区間に分割されて木に格納されているからです。

例えば区間 $[1, 7]$ は以下の区間から構成されます。



つまり次のように対応する区間を足せば良いです。

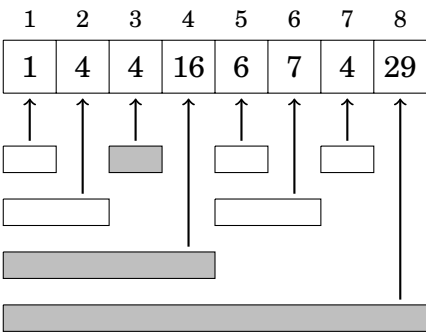
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

$\text{sum}_q(a, b)$ で $a > 1$ の値を計算するには、累積和のときと同じように求めます。

$$\text{sum}_q(a, b) = \text{sum}_q(1, b) - \text{sum}_q(1, a - 1).$$

$\text{sum}_q(1, b)$ と $\text{sum}_q(1, a - 1)$ の両方は $O(\log n)$ で計算できるのでこの時間計算量は $O(\log n)$ となります。

次は更新の操作を見ていきます。インデックス 3 の値が変化した場合には以下のノードに変化が生じます。



この操作も BIT では $O(\log n)$ 個の要素だけを更新すれば良いので更新の時間計算量は $O(\log n)$ であるとわかりました。

実装 - Implementation

BIT はビット演算を用いて効率的に実装することができます。 $p(k)$ の任意の値は次の式で計算で切るとというのが実装上のポイントです。

$$p(k) = k \& -k.$$

次の関数は $\text{sum}_q(1, k)$ の値を計算するものです。

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k&-k;
    }
    return s;
}
```

これを利用して位置 k の配列の値を x だけ増加させる関数は次のとおりです (x は正でも負でも良いです)。

```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k&-k;
    }
}
```

この操作は $O(\log n)$ です。BIT 内の $\log n$ 個の値にアクセスし、次の位置に移動するたびに $O(1)$ 時間の時間がかかるためです。(訳註: ここでは更新のみである値への設定については述べられていませんが、値を取得してからその値を引いて新しい値を加算すれば設定の操作となります)

9.3 セグメントツリー - Segment tree

セグメントツリー - segment tree^{*3} も区間クエリと値の更新をサポートするデータ構造です。セグメントツリーは和・最小・最大の問い合わせをはじめ、その他多くのクエリに対応して区間クエリと単一値の操作を $O(\log n)$ 時間で実行できるデータ構造です。

BIT と比較すると、セグメントツリーの利点はより一般的なデータ構造であることです。BIT は和の問い合わせのみをサポートします^{*4}。対してセグメントツリーは他のクエリもサポートします。ただし、より多くのメモリを必要として実装が複雑になります。

構造 - Structure

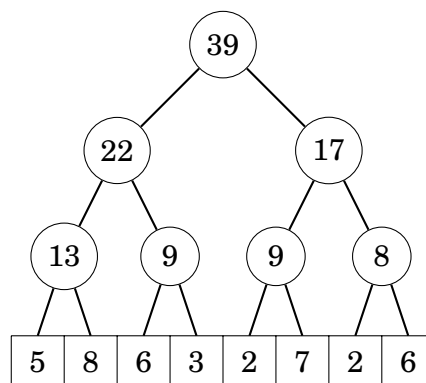
セグメントツリーは最下層のノードが配列要素に対応して、それより上位のノードは区間クエリに必要な情報を含む完全二分木の構造です。

ここでは配列のサイズが 2 の累乗でゼロベースでインデックスを作成することを想定します。基の配列のサイズが 2 の累乗でない場合は余分なノードを追加して 2 の累乗とします。

最初に和クエリに対応したセグメントツリーについて説明します。次のような配列を考えてみましょう。

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

これに対応するセグメントツリーは以下の通りです。



^{*3} The bottom-up-implementation in this chapter corresponds to that in [62]. Similar structures were used in late 1970's to solve geometric problems [9].

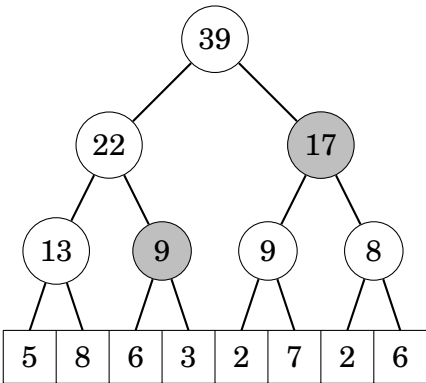
^{*4} In fact, using *two* binary indexed trees it is possible to support minimum queries [16], but this is more complicated than to use a segment tree.

内部 (訳註: 最下層より上の) 木の各ノードは、サイズが 2 のべき乗である配列の区間に対応しています。このツリーでは、各内部ノードの値は対応する配列の値の合計で、その左右の子ノードの値の合計として計算されています。

任意の区間 $[a, b]$ は、その値が木のノードに格納される $O(\log n)$ 個の区間に分割できることは明らかです。例えば区間 $[2, 7]$ を考えてます。

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

$\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$. について木の以下のノードに注目します。

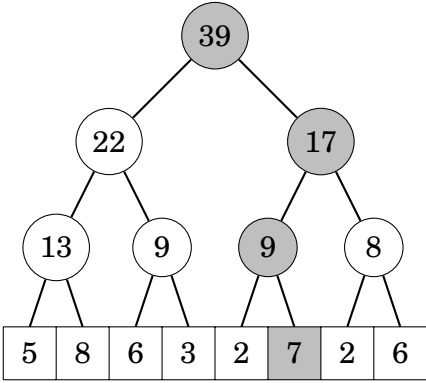


これを利用すると $9 + 17 = 26$ と求めることができます。

木のできるだけ高い位置にあるノードを用いて和を計算すると、同じ高さで計算に必要なノードは最大で 2 つ利用すれば良いです。従って処理する最大のノードの総数は $O(\log n)$ です。

それでは更新の操作を考えます。配列の更新を行った後には更新された値に依存する全ノードを更新する必要があります。更新された配列の要素から根のノードまでの経路に辿ったノードを更新すればよいです。

次の図は、配列の値 7 が変化した場合、どのツリーノードに更新が必要かを示しています。

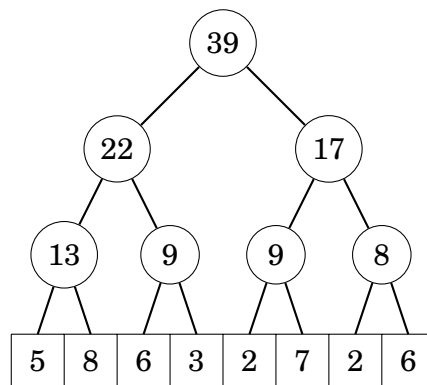


根からリーフノードへは $O(\log n)$ のノードで構成されるので、更新に必要な処理は $O(\log n)$ であることがわかります。

実装 - Implementation

セグメントツリーは、 $2n$ 個の要素からなる配列として格納されます。ここで n は元の配列のサイズであり、 2 の累乗とします。ツリーのノードは上から下に向かって格納されます。

$\text{tree}[1]$ は根となるノード。 $\text{tree}[2]$ と $\text{tree}[3]$ はその子というように構築します。 $\text{tree}[n]$ から $\text{tree}[2n-1]$ は元の値を表す最下段のノードとなります。



というセグメントツリーは次のような配列に格納ができます。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

この表現を用いると $\text{tree}[k]$ の親は $\text{tree}[\lfloor k/2 \rfloor]$ であり、その子は $\text{tree}[2k]$ と $\text{tree}[2k+1]$ と示せます。これはノードの位置が左の子であれば偶数、右の子であれば奇数であることを意味することを意味します。

次の関数は $\text{sum}_q(a, b)$ を計算します。

```

int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}

```

この関数の最初ではまず初期値として $[a+n, b+n]$ の区間に設定します。各ステップではノードの値を合計に追加して、その区間は木の 1 レベル上に移動します。

次に更新の操作をみてみましょう。次の関数はインデックス k の配列の値を x 増加させます。

```
void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k] + tree[2*k+1];
    }
}
```

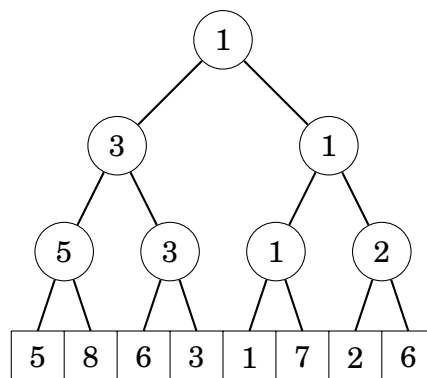
最初にツリーの最下層の値を更新します。その後は木の根に到達するまでのすべての内部ノードの値を更新していきます。

この関数は $O(\log n)$ で動作します。 n 個の要素を持つセグメントツリーは $O(\log n)$ 個のレベルから構成されており、各ステップでツリーの 1 レベル上を移動するためです。

和以外のクエリ - Other queries

セグメントツリーは区間を 2 つに分割してそれぞれについて別々に計算し、その答えを効率的に組み合わせることが可能な色々な区間クエリをサポートすることができます。このような他のクエリの例としては最小値と最大値・最大公約数・ビット演算の **and**、**or**、**xor** などが可能です。

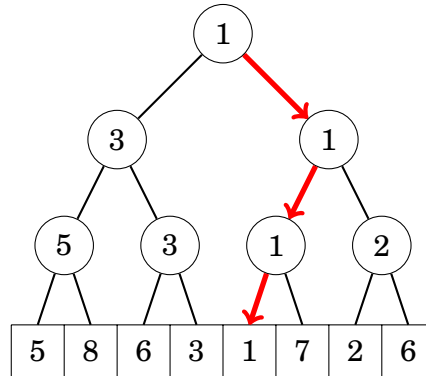
次のセグメントツリーは最小限のクエリーに対応します。



木のノードには対応する配列の区間内の最小値が含まれます。根のノードには配列全体の最小値が格納されることになります。この木は前述と同様に実装でき、和の代わりに最小値を計算します。

セグメントツリーの構造から配列要素の位置決めに二分探索を使用することができます。最小値に対応するツリーであれば最小値を持つ要素の位置を $O(\log n)$ 時間で見つけることができます。

上の木では、一番上のノードから下に向かってパスをたどることで、最小の値 1 を持つ要素を見つけることができます。



9.4 さらにテクニク - Additional techniques

座標圧縮 - Index compression

配列の上に構築されるデータ構造には留意すべき点があります。要素が連続した整数インデックスを持つことです。例えば、インデックス 10^9 を使いたい場合、配列には 10^9 個の要素が必要となるためメモリが不足してしまいます。このため大きなインデックスが必要な場合は工夫が必要になります。

元のインデックスをインデックス 1、2、3 などに置き換える **座標圧縮 - index compression** を利用することで、この制限を回避できます。これはアルゴリズムに必要なすべてのインデックスを事前に知っていれば簡単に実現できます。

このアイデアは元のインデックス x を $c(x)$ で置き換えるというもので c はインデックス番号を圧縮する関数です。この関数にはインデックスの順序が変わらないことを要求します。この関数があればインデックスが圧縮されていても簡単に問い合わせができます。

例えば、元のインデックスが 555, 10^9 , 8 の場合、新しいインデックスは次のようになります。

$$\begin{aligned}
 c(8) &= 1 \\
 c(555) &= 2 \\
 c(10^9) &= 3
 \end{aligned}$$

区間更新 - Range updates

これまで、区間クエリと単一の値の更新が可能なデータ構造を実装してきました。ここで区間の更新と単一の値の取得を行うことはできるでしょうか？ 区間 $[a, b]$ に含まれるすべての要素を x だけ増加させる操作を考えましょう。

実はこの章で紹介したデータ構造は、このような状況でも使うことができます。そのためには元の配列の連続した値の差分を示す**差分配列 - difference array**を構築します。つまり元の配列は差分配列の累積和の配列ということになります。次のような配列を考えてみましょう。

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

この差分配列を考えます。

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

元の配列の 6 の位置の値 2 は差分配列の和 $3 - 2 + 4 - 3 = 2$ に対応します。

差分配列の利点は区間を更新する際に差分配列の 2 つの要素を変更するだけで元の配列が完成する点です。元の配列の 1 から 4 までの値を 5 つ増やしたい場合は、差分配列の位置 1 の値を 5 増やしてインデックス 5 の値を 5 減らせばよいです。

こ以下のように示せます。

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

より一般的には区間 $[a, b]$ の値を x だけ増やすためには位置 a の値を x だけ増やし、位置 $b + 1$ の値を x だけ減らせばよいです。したがって、単一の値の更新と和クエリの処理だけが必要なので、BIT やセグメント木を使用して実装できます。

より困難な問題は区間クエリ可能な区間更新ですが、これについては第 28 章で取り上げましょう。

ここでビットの最初は符号に使われます。0 の場合は非負で 1 の場合は負を示します。そして、残りの $n-1$ ビットで数値が表現されます。負を表現するときには

応用 - Applications

$1 \ll k$ は k の位置に 1 ビットがあり他のビットはすべて 0 な整数を示すのである 1 ビットにアクセスすることができます。特に、 $x \& (1 \ll k)$ は 0 でないとき、数値の k 番目のビットが 1 だと判定できます。例えば、次のコードは、int 型の数値 x のビット表現を表示することが出来ます。

```
for (int i = 31; i >= 0; i--) {
    if (x & (1 << i)) cout << "1";
    else cout << "0";
}
```

また、同様に、数値の 1 ビットを変更することが可能です。例えば、 $x \mid (1 \ll k)$ は x の k ビット目を 1 にし、 $x \& \sim(1 \ll k)$

x の k ビット目を 0 にし $x \wedge (1 \ll k)$ は x の k 番目のビット目を反転させます。

$x \& (x - 1)$ は x の最後の 1 ビットを 0 にし、 $x \& -x$ は最後の 1 ビットを除き、すべての 1 ビットを 0 にします。 $x \mid (x - 1)$ は、最後の 1 ビット以降のビットをすべて反転させます。また、正の数 x は、 $x \& (x - 1) = 0$ のとき 2 の累乗です。

拡張機能 - Additional functions

C++ は以下のような組み込み関数を持っています。

- `__builtin_clz(x)`: 数値の先頭のゼロの個数
- `__builtin_ctz(x)`: 数値の末尾にあるゼロの個数
- `__builtin_popcount(x)`: 数値に含まれる 1 の数
- `__builtin_parity(x)`: 1 の数のパリティ（偶奇）

例を示します。

```
int x = 5328; // 000000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

これらは int のみに対応した関数です。long long に対応した関数も用意されており、最後に ll を付けて使用することができます。

10.3 集合の表現 - Representing sets

集合 $\{0, 1, 2, \dots, n-1\}$ のすべての部分集合は、1 ビットがどの要素が部分集合に属するかを示す n ビットの整数として表現することができます。各要素が 1 ビットのメモリしか必要とせず集合演算をビット演算として実装できるため、これは集合を表現する効率的な方法といえます。

例えば、`int` は 32 ビット型なので、`int` 型数値は集合 $\{0, 1, 2, \dots, 31\}$ の任意の部分集合を表現出来ます。集合 $\{1, 3, 4, 8\}$ のビット表現としては

000000000000000000000000100011010,

となり、 $2^8 + 2^4 + 2^3 + 2^1 = 282$ という数に対応します。

集合の実装 - Set implementation

以下のコードは `int` 型の x で表現できる $\{0, 1, 2, \dots, 31\}$ の操作を意味します。次のコードは 1, 3, 4, 8 を集合にに入れてそのサイズを表示しています。

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

そして、次のコードは、その集合に属するすべての要素を表示します

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

集合への操作 - Set operations

ビットを使って集合の操作を以下のように行うことが出来ます。

	set syntax	bit syntax
intersection	$a \cap b$	$a \& b$
union	$a \cup b$	$a \mid b$
complement	\bar{a}	$\sim a$
difference	$a \setminus b$	$a \& (\sim b)$

次の操作は $x = \{1, 3, 4, 8\}$ と $y = \{3, 6, 8, 9\}$ を作成したのちに、 $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$ を計算します。

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

反復作業 - Iterating through subsets

以下の操作は $\{0, 1, \dots, n-1\}$ の部分集合全てへの処理ができます。

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

ちょうど k 個の要素を含む部分集合に関する処理は次のように行えます。

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
}
```

次のコードは集合 x の部分集合を操作します。

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

10.4 Bit optimizations

多くのアルゴリズムで、ビット演算を利用して最適化することができます。このような最適化はアルゴリズムの時間計算量のオーダーを変えるものではありませんが、コードの実際の実行時間に大きな影響を与える可能性があります。このセクションでは、そのような状況の例について見ていきます。

ハミング距離 - Hamming distances

ハミング距離 - Hamming distance $\text{hamming}(a, b)$ は文字列 a, b の文字が異なる箇所の数です。例えば、以下の通りです。

$$\text{hamming}(01101, 11001) = 2.$$

長さ k の n 個のビット列のリストがあるとき、リスト中の 2 つの文字列間の最小のハミング距離を計算しなさいという問題を考えます。例えば、 $[00111, 01101, 11110]$ の答えは以下の通り 2 です。

- $\text{hamming}(00111, 01101) = 2$
- $\text{hamming}(00111, 11110) = 3$
- $\text{hamming}(01101, 11110) = 3$

この問題を解く簡単な方法は、すべての文字列のペアを調べて、そのハミング距離を計算することで、 $O(n^2k)$ で実行可能です。

ハミング距離の計算には以下の関数を用いることができます。

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

しかし、 k が小さい場合は、ビット列を整数として格納し、ビット演算でハミング距離を計算することで、コードを最適化することができます。 $k \leq 32$ と仮定すると文字列を `int` として格納し、以下の関数で距離を計算できます。

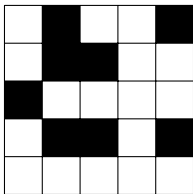
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

上記の関数では、`xor` 演算により、 a, b が異なる位置に 1 ビットを持つビット列が構成されます。そして、そのビット数を `__builtin_popcount` 関数でカウントします。

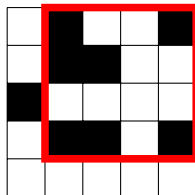
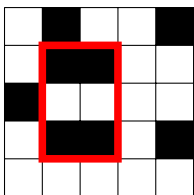
実装を比較するために、長さ 30 のランダムなビット列 10000 個のリストを生成した。最初のアプローチでは、検索に 13.5 秒かかりましたが、ビット最適化した関数のアプローチでは 0.5 秒しかかかりませんでした。このように、ビット最適化後のコードは、元のコードに比べて約 30 倍高速になりました

サブグリッドのカウント - Counting subgrids

各マスが黒(1)か白(0)である $n \times n$ 個のグリッドがあるときに、すべての角が黒であるサブグリッドの数を計算する問題を考えます。



次のようなサブグリッドが存在します。



この問題を解く $O(n^3)$ 時間のアルゴリズムは簡単です。行のすべての $O(n^2)$ 組を調べ、各組 (a, b) について、両方の行で黒マスを含む列の数を $O(n)$ 時間で算出します。以下のコードでは、`color[y][x]` が y 行 x 列の色を表すと仮定しています。

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) count++;
}
```

そして、`count(count-1)/2` 個の黒い角を持つサブグリッドが答えです。サブグリッドを形成するために、そのうちの任意の 2 個を選んでいるためです。

このアルゴリズムを最適化するために、グリッドを列のブロックに分割し、各ブロックが連続する N 個の列で構成されるようにしましょう。そして、各列は、正方形の色を表す N ビット数のリストとして格納します。これで、 N 個の列をビット演算で同時に処理できるようになりました。以下のコードでは、`color[y][k]` が N 個の色のブロックをビットで表しています。

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

これは $O(n^3/N)$ で動作します。

サイズ 2500×2500 のランダムなグリッドを生成し、オリジナルとビット最適化

された実装を比較しました。オリジナルのコードが 29.6 秒かかったのに対し、ビット最適化版は $N = 32$ (int 数) で 3.1 秒、 $N = 64$ (long 長数) で 1.7 秒で済みました。

10.5 動的計画法へのビット演算の利用 - Dynamic programming

ビット演算は動的計画法においても要素の部分集合を含む状態を整数で保存できるため、効率的で便利な実装出来ることがあります。ビット演算と動的計画法の組み合わせの例について見ていきましょう。

最適な選択 - Optimal selection

最初の例として、次の問題を考えます。 n 日間にわたる k 個の商品の価格が与えられており、各商品をちょうど 1 回買いたい。しかし 1 日に買うことができるのは 1 個までである。最小の合計金額は？ 例えば、次のようなシナリオを考えることにします。($k = 3, n = 8$).

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

このシナリオでは 5 が答えです。

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

例えば、上のシナリオでは $\text{price}[2][3] = 7$ です。この関数を用いると、問題の解は $\text{total}(0 \dots k - 1, n - 1)$ となります。空集合を買うにはコストがかからないので $\text{total}(\emptyset, d) = 0$ 、初日に商品を 1 つ買う方法は 1 つなので $\text{total}(x, 0) = \text{price}[x][0]$ として良いです。そうすると、次のような漸化式を利用できます。

ここでは $\text{price}[x][d]$ は d での x の価格を示すとしましょう。例えば、上のシナリオでは $\text{price}[2][3] = 7$ です。 $\text{total}(S, d)$ を、 d 日目時点での部分集合 S を買うための最小のコストとしましょう。これを用いると $\text{total}(\{0 \dots k - 1\}, n - 1)$ が求めたい値となります。

まず、空集合を買うにはコストがかからないので $\text{total}(\emptyset, d) = 0$ として良いで

す。初日に商品を1つ買う方法は1つなので $\text{total}(\{x\}, 0) = \text{price}[x][0]$ です。そうすると、次のような漸化式となります。

$$\text{total}(S, d) = \min(\text{total}(S, d-1), \min_{x \in S} (\text{total}(S \setminus x, d-1) + \text{price}[x][d]))$$

これは、 d 日にどの商品も買わないか、 S に属する商品 x を買うかのどちらかを意味します。後者の場合、 x を S から外し、 x の価格を合計価格に加算します。

では動的計画法で関数の値を計算するための配列を用意しましょう。

```
int total[1<<K][N];
```

ここで、 K と N は適当な大きさの定数です。配列の最初の次元は、部分集合のビット表現に対応します。

まず、 $d = 0$ の場合は、以下のように処理することができます。

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

そして次のように書くことができます。

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s & (1<<x)) {
                total[s][d] = min(total[s][d],
                                   total[s^(1<<x)][d-1] + price[x][d]);
            }
        }
    }
}
```

これは $O(n2^k k)$ で動作します。

順列からの集合 - From permutations to subsets

動的計画法を用いると、順列の繰り返しを部分集合の繰り返しに変更することができる場合があります。^{*1} 順列の組み合わせ数である $n!$ はとても大きな数となり、部分集合の組み合わせ数である 2^n よりもずっと大きくなるので非常に有効な方法

^{*1} This technique was introduced in 1962 by M. Held and R. M. Karp [34].

です。たとえば、 $n = 20$ として $n! \approx 2.4 \cdot 10^{18}$ であり、 $2^n \approx 10^6$ です。したがって、ある種の n の値に対して、部分集合は効率的に調べられるが、順列を調べることは困難です。

最大重量 x のエレベーターがあり、1 階から最上階まで行きたい体重が既知の n 人がいるとする問題を考えます。人々が最適な順序でエレベーターに乗り込む場合、必要な最小の乗車回数は何回ですか？

例えば、 $x = 10$ 、 $n = 5$ で、重みは以下の通りだとしましょう。

person	weight
0	2
1	3
2	3
3	5
4	6

この場合、最小の解は 2 です。最適な順序の 1 つは $\{0, 2, 3, 1, 4\}$ で、これは人々を 2 分割し、最初に $\{0, 2, 3\}$ (合計重量 10)、次に $\{1, 4\}$ (合計重量 9) で運べば良いです。

この問題は、 n 人の可能な順列をすべてテストすることにより、 $O(n!n)$ 時間で簡単に解くことができます。しかし、動的計画法を使って、より効率的な $O(2^n n)$ 時間のアルゴリズムを得ることもできます。そのアイデアは、人の各サブセットについて、必要な最小乗車数と最後のグループで乗車する人々の最小重量という 2 つの値を計算することです。

ここで、 $\text{weight}[p]$ を人物 p の重みとしましょう。また、部分集合 S の最小乗車数を $\text{rides}(S)$ 、最後の乗車の重みを $\text{last}(S)$ とします。

例えば、上記のシナリオの場合は以下ようになります。

$$\text{rides}(\{1, 3, 4\}) = 2 \quad \text{かつ} \quad \text{last}(\{1, 3, 4\}) = 5,$$

なぜなら最適な乗り物は $\{1, 4\}$ と $\{3\}$ であり、2 番目の乗り物は重み 5 だからである。もちろん、我々の最終目標は $\text{rides}(\{0 \dots n-1\})$ の値を計算することです。

関数の値を再帰的に計算し、動的計画法を適用すればよいです。これは、 S に属するすべての人を調べて、最後にエレベーターに入る人 p を最適に選ぶというアルゴリズムです。そのような選択のたびに、より小さな人々の部分集合に対する部分問題が生じます。 $\text{last}(S \setminus p) + \text{weight}[p] \leq x$ ならば、 p を最後に乗る人に加えることができます。そうでなければ、最初は p しか乗っていない新しいカゴを確保しないといけません。動的計画法を実装するために、次のような配列を用意します。

```
pair<int, int> best[1<<N];
```

各分割集合 S に対して $(rides(S), last(S))$ のペアを含みます。空のグループに対する値を以下のように設定します。

```
best[0] = {1,0};
```

そして次のように計算ができます。

```
for (int s = 1; s < (1<<n); s++) {
    // initial value: n+1 rides are needed
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto option = best[s^(1<<p)];
            if (option.second+weight[p] <= x) {
                // add p to an existing ride
                option.second += weight[p];
            } else {
                // reserve a new ride for p
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

このループは、 S_1 and S_2 のような任意の 2 つの部分集合 S_1 と S_2 に対して、 S_2 の前に S_1 を処理することを保証できます。動的計画法は正しい順序で値を計算されます。

サブセットのカウント Counting subsets

最後の問題です。 $X = \{0 \dots n-1\}$ として $S \subset X$ ここで各サブセットには $value[S]$ の価値割り当てられているとします。私たちのタスクは全ての S に対して以下を計算することです。

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A],$$

つまり、 S の部分集合の価値総和です。

例えば、 $n = 3$ の時、以下のように値を割り当てます。

- $value[\emptyset] = 3$
- $value[\{0\}] = 1$
- $value[\{1\}] = 4$
- $value[\{0, 1\}] = 5$

- $\text{value}[\{2\}] = 5$
- $\text{value}[\{0,2\}] = 1$
- $\text{value}[\{1,2\}] = 3$
- $\text{value}[\{0,1,2\}] = 3$

このケースでは、以下のように計算できます。

$$\begin{aligned}\text{sum}(\{0,2\}) &= \text{value}[\emptyset] + \text{value}[\{0\}] + \text{value}[\{2\}] + \text{value}[\{0,2\}] \\ &= 3 + 1 + 5 + 1 = 10.\end{aligned}$$

サブセットは全部 2^n あるので、1つの可能な解決策は、サブセットのすべてのペアを $O(2^{2n})$ 時間で処理することです。動的計画法を用いると、 $O(2^n n)$ 時間で解くことができます。このアイデアは、 S から削除される可能性のある要素が制限されているということです。

例えば、 S から要素 $0 \dots k$ だけを取り除いてもよいという制限のある S の部分集合の値の総和を $\text{partial}(S, k)$ とする。ここで、 $\text{partial}(S, k)$ を、集合 S から $0 \dots k$ 取り除いて良いものとしましょう。

$$\text{partial}(\{0,2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0,2\}],$$

というのは、要素 $0 \dots 1$ だけを削除することができるからですここで、 sum は partial を使って表現できます。

$$\text{sum}(S) = \text{partial}(S, n-1).$$

初期値は

$$\text{partial}(S, -1) = \text{value}[S],$$

とします。この場合、 S から要素を削除することはできないからです。一般的な場合、以下の再帰で表現できます。

$$\text{partial}(S, k) = \begin{cases} \text{partial}(S, k-1) & k \notin S \\ \text{partial}(S, k-1) + \text{partial}(S \setminus \{k\}, k-1) & k \in S \end{cases}$$

ここで k に注目します。 $k \in S$ の場合、 S から k 消すか残すかを選択することが出来ます。

ここで和の計算を実装するのに特に賢い方法があります。

```
int sum[1<<N];
```

各サブセットの合計を含むことになります。配列は以下のように初期化されます。

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

次のように処理を行います。

```
for (int k = 0; k < n; k++) {  
    for (int s = 0; s < (1<<n); s++) {  
        if (s&(1<<k)) sum[s] += sum[s^(1<<k)];  
    }  
}
```

このコードは、 $k = 0 \dots n-1$ の $\text{partial}(S, k)$ の値を配列の和に計算します。 $\text{partial}(S, k)$ は常に $\text{partial}(S, k-1)$ に基づいているので、配列 `sum` を再利用でき、非常に効率的に動作します。

第 II 部

グラフアルゴリズム - Graph algorithms

第 11 章

グラフの基礎知識 - Basics of graphs

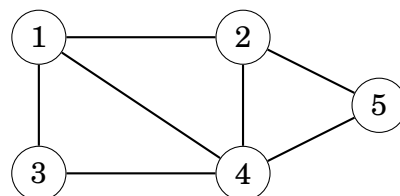
競技プログラミング問題のいくつかは、問題をグラフの問題としてモデル化して適切なアルゴリズムを用いることで解決することができます。グラフの典型的な例として、街を道路でつなぐネットワークとみなして最短の経路を求める問題などあります。しかし、コンテストではグラフが問題の中に隠れていてグラフを用いるという発見が難しいことも多々あります。

このパートでは、特に競技プログラミングで重要となるトピックを中心に解説します。この章では、グラフに関する概念を述べ、実装で必要になるグラフのさまざまな表現方法について述べます。

11.1 グラフの用語 - Graph terminolog

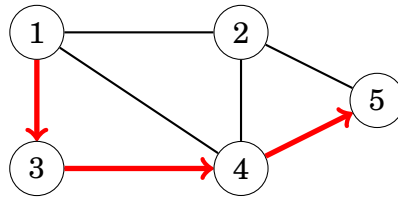
グラフ - **graph** はノード - **node** と辺 - **edge** から構成されます。本書では以降、変数 n はグラフのノード数、変数 m は辺の数を表します。ノードには $1, 2, v, \dots, n$ の整数を用いて表します (訳註: 0-indexed ではありません)。

例えば、次のグラフは 5 つのノードと 7 つのエッジから構成されています。



ノード a からノード b まで、グラフの辺を通る経路 (パス - **path**) があります。パスの長さは、その中に含まれる辺の数とします。例えば、上のグラフにはノード 1

からノード 5 まで長さ 3 のパス $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ が存在します。

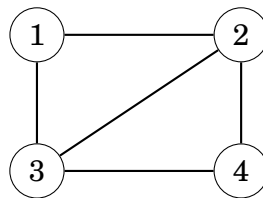


最初と最後のノードが同じものは**閉路 (サイクル, Cycle)** と呼ばれます。例えば、上のグラフは $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ という閉路を含みます。

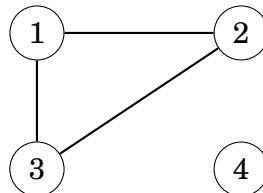
各ノードの出現回数がせいぜい 1 回であるパスは**単純**と呼ばれます。

連結 - Connectivity

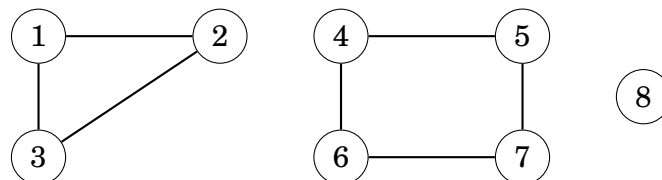
グラフは、ある任意の 2 つのノード間にパスが存在する場合、**連結**といえます。例えば、次のようなグラフは連結と呼ばれます。



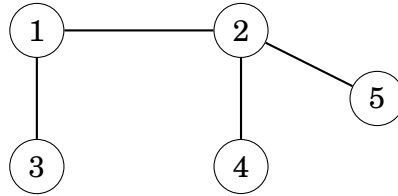
次のグラフは、ノード 4 から他のノードに行くことができないので、連結ではないグラフです (非連結)。



グラフの各連結な部分を**成分 (components)** を成分と呼びます。例えば、次のグラフは 3 つの成分を含んでいます。{1, 2, 3}, {4, 5, 6, 7}, {8} です。

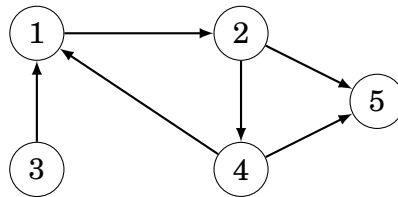


木 - tree は、 n 個のノードと $n-1$ 個の辺からなる連結グラフのことです。木の任意の 2 つのノード間にはちょうど一本のパスが存在します。例えば、次のグラフは木である。



有向と無向 - Edge directions

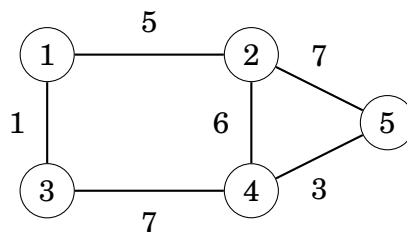
グラフの辺が決まった方向にしか移動できない場合、**有向 (directed)** と呼ばれます。例えば、次のようなグラフは有向です。



このグラフには、ノード 3 からノード 5 へのパス $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ がありますが、ノード 5 からノード 3 へのパスは存在しません。

辺の重み - Edge weights

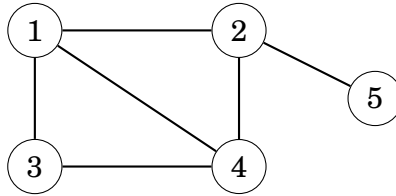
重み付きグラフ - weighted graph では、各辺に**重み**が割り当てられています。重みは辺の長さのようなものです。重み付きグラフの例を次に示します。



重み付きグラフのパスの長さは、パス上の辺の重みの和で表します。例えば、上のグラフでは、 $1 \rightarrow 2 \rightarrow 5$ のパスの長さは 12。 $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ のパスの長さは 11 です。後者の経路はこのグラフのノード 1 からノード 5 への**最短経路**です。

隣接ノードと次数 - Neighbors and degrees

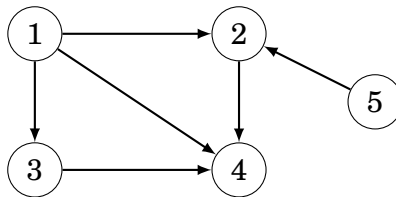
2つのノードは、その間にエッジがある場合、**隣接 (neighbors, adjacent)** であるといいます。**次数 (degree)** は、ノードの隣接するノードの数です。例えば、次のグラフでは、ノード 2 の近傍は 1、4、5 なので、その次数は 3 です。



グラフに含まれるノードの次数の和は常に $2m$ (m は辺の数) となります。なぜなら、各辺はちょうど 2 つのノードの度数を 1 つずつ増加させるからです。このため、次数の和は常に偶数となります。

グラフは、すべてのノードの次数が同じ (例えば定数 d) であれば**正則 (regular)** グラフです。すべてのノードの次数が $n-1$ であれば、すなわちグラフがノード間の可能なすべての辺を含んでいれば、**完全 (complete)** グラフです。

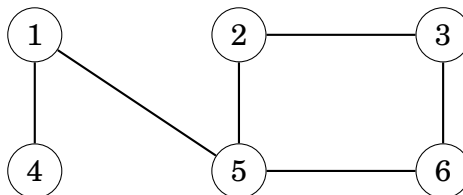
有向グラフにおいて各ノードには入次数と出次数があります。**入次数 (indegree)** とはそのノードで終わる辺の数、**出次数 (outdegree)** そのノードで始まる辺の数です。例えば、次のグラフでは、ノード 2 の入次数は 2、ノード 2 の出次数は 1 です。



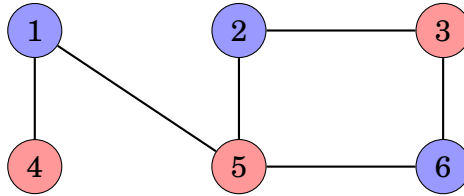
グラフの着色 - Colorings

グラフの**彩色 (coloring)** は、隣接するノードが同じ色にならないように、各ノードに色を割り当てることをいいます。グラフを 2 色で色付けできる場合、**二部グラフ (bipartite graph)** であるといえます。なお、二部グラフである条件として奇数本の辺で構成される閉路が存在しないグラフに限られることが分かっています。

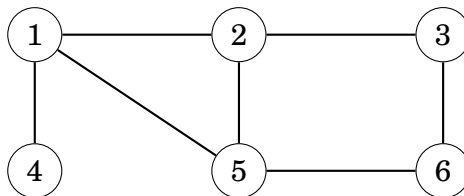
例を示します。



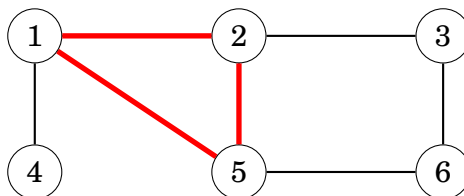
このグラフは次のように着色でき、二部グラフです。



次のグラフはどうでしょうか？

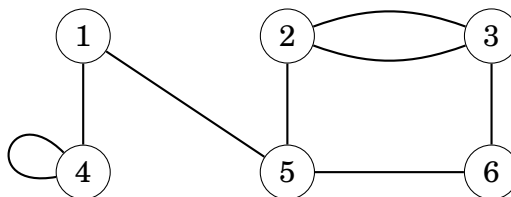


は 2 分割ではありません。なぜなら、以下の閉路を 2 色で色塗りできないためです。



単純グラフ - Simplicity

始点と終点が同じノードの辺がなく、ある 2 つのノード間に複数の辺が存在しない場合、グラフは**単純グラフ (simple graph)**と呼ばれます。大半の場合、グラフは単純だと仮定されます。例えば、次のようなグラフは単純ではありません。



11.2 グラフの表現方法 - Graph representation

実装する上でグラフを表現する方法はいくつかあります。どのようなデータ構造を使うというのはグラフの大きさやアルゴリズムの実装方法に依存します。一般的な 3 つの表現方法を説明する。

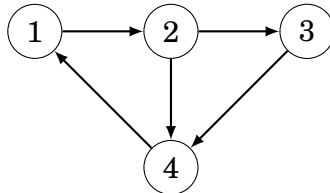
隣接リスト形式 - Adjacency list representation

隣接リストでは、グラフの各ノード x に、 x から張られている辺の先をリストで持ちます。これは非常に一般的な表現手法で、ほとんどのアルゴリズムはこれを

用いて効率的に実装できます。隣接リストを格納する便利な方法は、次のように `vector` の配列を宣言します。

```
vector<int> adj[N];
```

定数 N はグラフのノードの数です。例えば次のグラフを考えます。



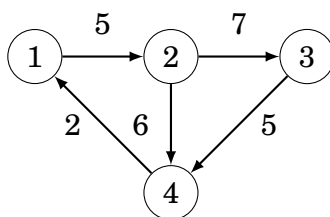
これは隣接リストで次のように表現できます。

```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

グラフが無向の場合は両方から互いに対して辺を張ることで表現できます。
重み付きグラフの場合は少し持たせ方を変えます。

```
vector<pair<int,int>> adj[N];
```

このとき、ノード a の隣接リストには、ノード a からノード b へ重み w の辺があるとき、 (b, w) の `pair` を作ります。例えば、次の例を考えます。



これは次のように表現できます。

```
adj[1].push_back({2, 5});
adj[2].push_back({3, 7});
adj[2].push_back({4, 6});
adj[3].push_back({4, 5});
adj[4].push_back({1, 2});
```

隣接リストを用いる利点は、あるノードから辺を経由して移動できるノードを効

率的に見つけられることです。例えば、以下の `for` 文で、ノード s から移動できるすべてのノードを処理できます。

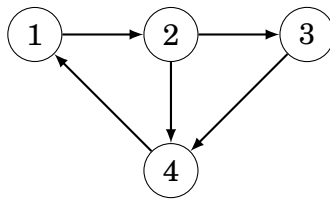
```
for (auto u : adj[s]) {
    // process node u
}
```

隣接行列形式 - Adjacency matrix representation

隣接行列は、辺を 2 次元の配列で表現します。2 つのノード間にエッジがあるかどうかを効率的に調べることができます。

```
int adj[N][N];
```

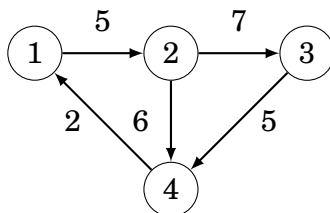
ここで、 $adj[a][b]$ は、グラフにノード a からノード b への辺が含まれるかをしめし、 $adj[a][b] = 1$ なら辺があることを、 $adj[a][b] = 0$ なら辺がないことを示します。例えば次のグラフを考えます。



次のように示せます。

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

辺に重みがある場合は隣接行列の表現を拡張します。 $adj[a][b]$ の値としてエッジの重みが含まれるようにします。例を示します。



これは次のように表現できます。

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

隣接行列表現の欠点として行列が n^2 要素分のメモリを必要とし、通常はそのほとんどが 0 となってしまうためです。このため、グラフが大きい場合はこの表現は適しません。

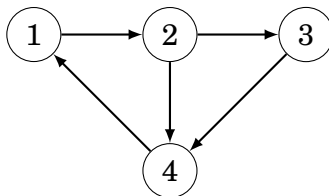
エッジリスト形式 - Edge list representation

エッジリスト形式は、全ての辺を順序通りに保管したものです。アルゴリズムがグラフのすべてのエッジを処理するのみに便利です。ただし、あるノードから始まるエッジを見つける必要がない場合は適しません。

次のように情報を持たせます。

```
vector<pair<int,int>> edges;
```

(a,b) に辺があることを示します。例を示します。



これは次のように表現できます。

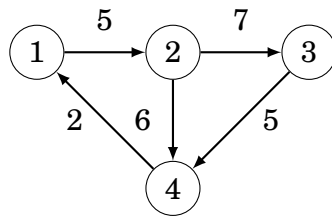
```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

グラフに重みがある場合、以下のように持たせることができます。

```
vector<tuple<int,int,int>> edges;
```

各要素は (a,b,w) であり、ノード a からノード b へ重み w の辺があることを意味

します。例を示します。



これは次のように表現できます。^{*1}:

```
edges.push_back({1,2,5});  
edges.push_back({2,3,7});  
edges.push_back({2,4,6});  
edges.push_back({3,4,5});  
edges.push_back({4,1,2});
```

^{*1} In some older compilers, the function `make_tuple` must be used instead of the braces (for example, `make_tuple(1,2,5)` instead of `{1,2,5}`).

第 12 章

グラフ探索 - Graph traversal

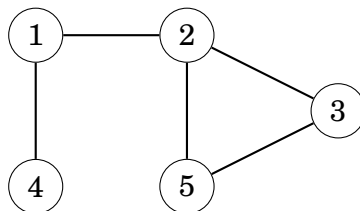
この章では、深さ優先探索 (DFS) と幅優先探索 (BFS) という 2 つの基本的なグラフアルゴリズムについて説明します。どちらのアルゴリズムもグラフの開始ノードが与えられ、その開始ノードからすべての到達可能なノードを訪問し、両者はこの訪問順序が異なります。

12.1 深さ優先探索 - Depth-first search

深さ優先探索 (Depth-first search) (DFS) は直線的なグラフ探索技法です。開始ノードからエッジを使用して到達可能な他のすべてのノードに進みます。深さ優先探索は、新しいノードがある限り常にグラフ内の下側に辿ります。行き止まりになったら、前のノードに戻りグラフの他の部分の探索を開始します。このアルゴリズムは、訪問したノードを記録しておき各ノードを一度だけ処理するようにします。

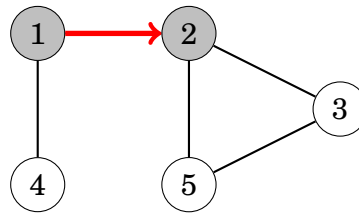
例

次のグラフを深さ優先探索で処理する方法を考えます。

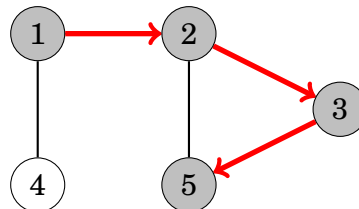


グラフのどのノードから検索を始めても良いですが、ここではノード 1 から検索を始めます。

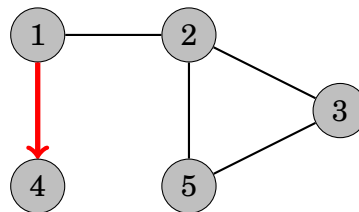
まず、ノード 2 を探索します。



そして、3,5 と訪問したとします。



ノード 5 の隣接ノードは 2 と 3 ですが、すでにその両方を訪れているので、前のノードに戻ります。また、ノード 3 と 2 の隣接ノードは全て訪問済みなので、次はノード 1 からノード 4 へ移動することになります。



これですべてのノードを訪問したので探索は終了します。

深さ優先探索の時間計算量は $O(n+m)$ (n はノードの数、 m はエッジの数) です。なぜなら全てのノードと辺が1回ずつ辿るからです。

DFS の実装

深さ優先探索は再帰を使って簡単に実装できます。次の関数 `dfs` は、引数に与えられたノードから深さ優先探索を開始します。この関数は、グラフを隣接リストとして参照します。

```
vector<int> adj[N];
```

また、次の配列を訪問済みのノードの情報として利用します。

```
bool visited[N];
```

訪問済みリストの値は最初全て `false` です。そして、`s` でこの関数が呼ばれると `visited[s]` は `true` になります。この関数は以下のように実装できます。

```

void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: adj[s]) {
        dfs(u);
    }
}

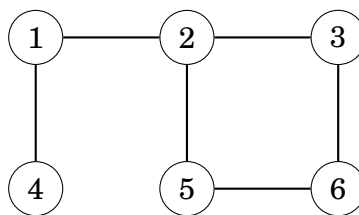
```

12.2 幅優先探索 - Breadth-first search

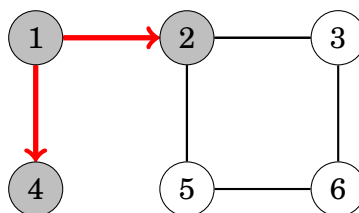
幅優先探索 - Breadth-first search (BFS) は、開始ノードからの距離順が小さい順にノードを訪問していきます。つまり、**BFS** を用いると始点ノードから他のすべてのノードまでの距離を計算することができます (訳註: これは **DFS** でも可能です)。ただし、**BFS** は深さ優先探索よりも実装が複雑になります。**BFS** はノードを深さごとに見ていきます。まず、開始ノードからの距離が1であるノードを探索し、次に距離が2であるノードを探索し、というようにすべてのノードが訪問されるまで続けられます。

例

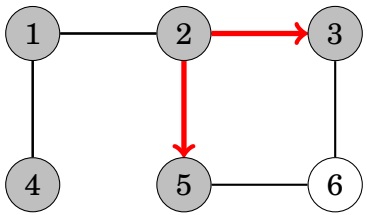
次のようなグラフに対して、幅優先探索がどのように処理されるかを考えてみよう。



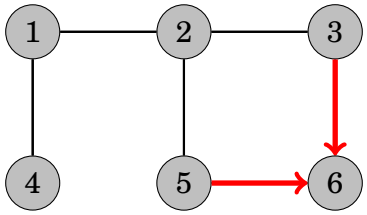
ノード1から探索を開始したとします。最初に、ノード1から直接つながっている1つの辺で到達可能なすべてのノードを探索します。



その後、ノード3、ノード5を探索します。



最後に、ノード 6 が探索されます。



開始ノードからグラフの全ノードまでの距離を以下のように計算できました。

node	distance
1	0
2	1
3	2
4	1
5	2
6	3

BFS も DFS と同様に $O(n + m)$ の計算量で実行できます。先ほどと同様に n がノードの数で m が辺の数とします。

実装

BFS は今探索しているノードとは異なる部分のノードを訪問するため、DFS よりも実装が少し複雑になります。最もシンプルなアプローチは探索するノードのキューを持ち、各ステップではキュー内の最初のノードを処理します。

以下のコードはグラフが隣接リストとして格納され、以下の変数を保持することを想定しています。

```
queue<int> q;
bool visited[N];
int distance[N];
```

キュー q には処理すべきノードが距離の昇順に並んでいます。新しいノードは常

にキューの末尾に追加されて待ち行列の先頭にあるノードが次に処理されるノードとなる。配列 `visited` は訪問済みのノードであるかを持ち、配列 `distance` は開始ノードからグラフの全ノードまでの距離を持ちます。

ノード `x` から始まる探索は、以下のように実装できます。

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // process node s
    for (auto u : adj[s]) {
        if (visited[u]) continue;
        visited[u] = true;
        distance[u] = distance[s]+1;
        q.push(u);
    }
}
```

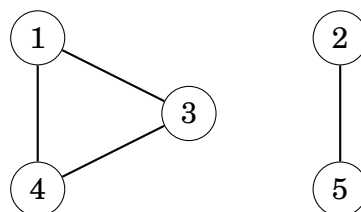
12.3 応用 - Applications

グラフの探索アルゴリズムを用いると、いくつかのグラフの特性を確認することができます。多くのケースで **DFS**, **BFS** の両方を用いることができますが、実装が簡易性から **DFS** を選択することが多いでしょう。以下の応用例では、グラフは無向であると仮定する。

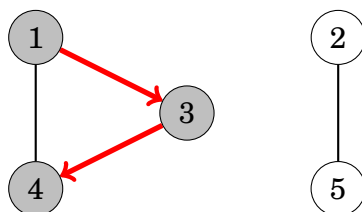
連結性チェック - Connectivity check

グラフの任意の 2 つのノード間にパスが存在するとき、グラフは連結していると言えます。つまり任意のノードから出発して、他のすべてのノードに到達できるかどうかを調べれば、グラフが連結されているかどうかを調べることができます。

例えば次のようなグラフを考えます。



DFS をノード 1 から実行します。

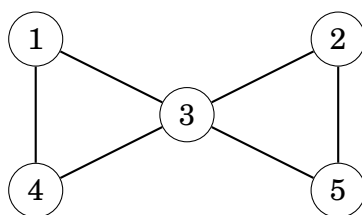


この結果、すべてのノードが訪問されなかったので、このグラフは連結されていないと結論づけられます。このあと、探索されていないノードからさらに新しい深さ優先探索を開始することにより、グラフのすべての連結成分を見つけることもできます。

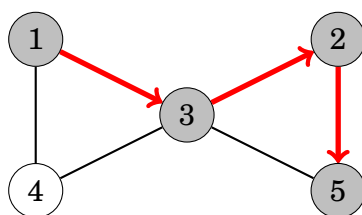
閉路の検出 - Finding cycles

あるノードの探索中に、(現在のパスの前のノード以外の) その隣接ノードがすでに訪問されている場合、グラフは閉路を含んでいることになります。

これも例で示します。



これには2つの閉路が含まれます。例えばこの1つを見つけるのは以下のように行われます。



ノード 2 からノード 5 に移動した後、ノード 5 の隣接ノード 3 はすでに訪問済みであるとわかります。したがって、このグラフには、例えば $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$ のような 3 を含む閉路があるとわかります。

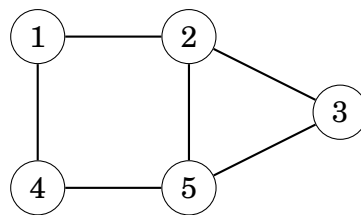
グラフが閉路を含むかどうかを調べるにはもう 1 つ方法があります。各要素のノードとエッジの数を数えます。 c 個のノードの連結のグラフは閉路がなければちょうど $c-1$ 本のエッジを含むはずですが (つまり、木でなければならない)。 c 個以上の辺があれば、この連結成分には必ず閉路が含まれます。

二部グラフチェック - Bipartiteness check

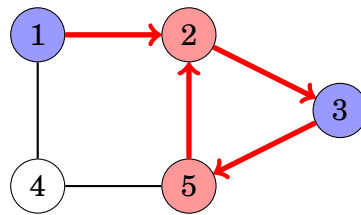
グラフを同じ色のノードが隣接しないようにノードが2色で着色できる場合二部グラフと呼ばれます。グラフの探索アルゴリズムを用いて二部グラフかどうかを調べるのは非常に簡単です。

例えば、開始ノードを青、その隣をすべて赤、その隣をすべて青、といった具合に色分けしていきます。探索のある時点で、隣接する2つのノードが同じ色であることに気づいたら、そのグラフは二部グラフではありません。そうでなければ、グラフは二部グラフにでき、その1つの色付けが見つけたことになります。

例えば以下のグラフがあります。



これはノード1からの探索が次のようになるので条件を満たしません。



隣接するノード2と5は共に赤です。したがって、このグラフは二部グラフにはできません。

利用可能な色が2色しかない場合、コンポーネントの開始ノードの色がそのコンポーネントの他のすべてのノードの色を決定します。開始ノードが赤であろうと青であろうと、結果は変わりません。

ここで注意があります。グラフのノードを k 色で着色して、隣接するノードが同じ色にならないようにできるかどうかを調べることは非常に困難です。 $k=3$ の場合でも、効率的なアルゴリズムは知られておらず、この問題はNP困難 (*NP-hard*) です。

第 13 章

最短経路

グラフの 2 ノード間の最短経路を求めることは重要なトピックであり様々な応用が可能です。複数の都市があって各道路の長さが与えられたときに、ある 2 つの都市を結ぶ路線の最短距離を計算する、などです。重みのないグラフでは、パスの長さはその辺の数に等しいので、単純な幅優先探索 (BFS) で最短経路を求めることができます。

この章では、重み付きグラフで最短経路を見つけるための洗練されたアルゴリズムをみていきます。

13.1 最短経路 (Bellman – Ford)

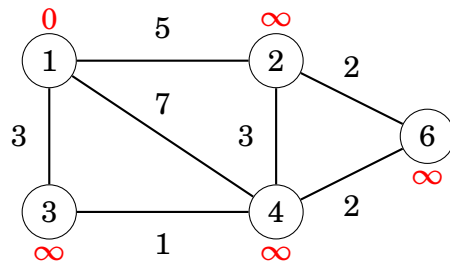
Bellman – Ford algorithm^{*1} はあるノードから開始して全てのノードへの最短経路を計算します。このアルゴリズムは負のコストとなる閉路を持たない全てのグラフで利用できます。負のコストとなる閉路が存在する場合はそれを検出します。

このアルゴリズムは、開始ノードからすべてのノードまでの距離を調べます。まず、開始ノードまでの距離は 0 であり、他のすべてのノードまでの距離は無限とします。そして、どの距離も小さい距離に更新できなくなるまで、より小さな距離に更新できる辺を見つけていきます。

Example

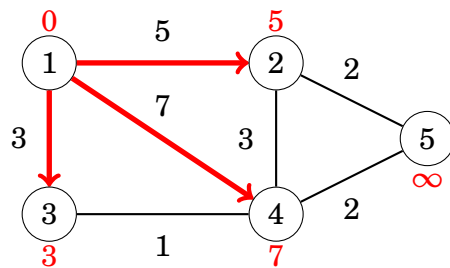
次の図で Bellman – Ford アルゴリズムの動作を説明します。

^{*1} The algorithm is named after R. E. Bellman and L. R. Ford who published it independently in 1958 and 1956, respectively [5, 24].

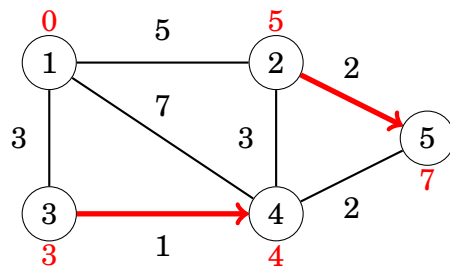


各ノードは始点からの距離を持ちます。初期状態では開始ノードの距離は 0 で、それ以外の距離は INF です。

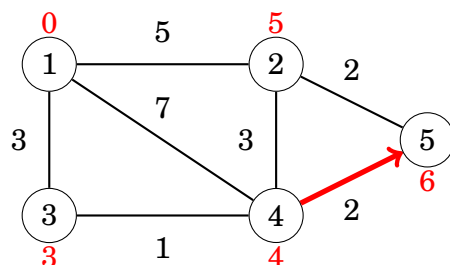
このアルゴリズムでは、あるノードから隣接するノードの距離を縮めることのできる辺を調べます。ノード 1 から始めるとしましょう。全ての隣接ノードへの距離は (初期値が INF なので) 縮まります。



2 → 5 と 3 → 4 の 2 つの辺が距離を縮めます。

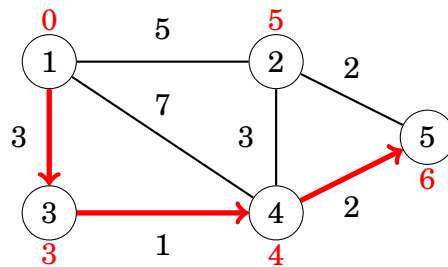


最後にもう 1 つ更新できます。



これ以上は距離を縮めることはできません。これは開始ノードからの最短距離が確定したことを意味します。

例えば、ノード 1 からノード 5 までの最短距離 3 は、次のような経路となりました。



実装

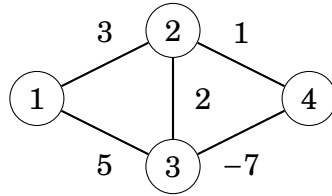
ノード x からグラフの全ノードまでの最短距離を求める実装を以下に示します。この実装は、グラフが (a, b, w) というタプルのリスト `edges` で表現されているとします。それぞれの要素は辺を表し、ノード a からノード b 向きに重み w の辺があるとしましょう。Bellman-Ford は $n-1$ ラウンドで構成されます。各ラウンドはグラフのすべてのエッジを調べて隣接ノードへの距離を縮めようと試みます。始点 x からグラフの全ノードまでの距離を格納する配列 `distance` を定義します。なお、定数 `INF` は無限遠 (十分に大きな数) を表します。

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

時間計算量を考えると、先に述べた通り、 $n-1$ 回のラウンドで構成され、各ラウンドで m 個の辺をすべて繰り返し処理するので $O(nm)$ となります。グラフに負の閉路が存在しない場合、最も長い最短パスが $n-1$ の辺で構成されていても、 $n-1$ 回のラウンドで確定します。多くの場合は $n-1$ ラウンドよりも早く求められるのが普通でしょう。つまり、あるラウンドで距離を縮めることができなければアルゴリズムを打ち切って良いです。

負の閉路 - Negative cycles

また、Bellman - Ford アルゴリズムは、グラフに負の閉路が含まれているかどうかを確認するために使用することができます。以下に例を示します。



グラフに和が -4 となる負の閉路 $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ があります。このようにグラフに負の閉路がある場合、無限にその閉路に関係するノードの距離が小さくなっていきます。すなわち、最短経路という概念は意味をなさないとはいえます。負の閉路は、*Bellman-Ford* を使用して、アルゴリズムを n ラウンド実行することによって検出することができます (訳註: $n-1$ ではないことに注意)。この追加ラウンドで距離が縮まればそのグラフは負の閉路を含むと判定されます。これを用いて、グラフ上の負の閉路を探索することができます。

SPFA アルゴリズム - SPFA algorithm

SPFA アルゴリズム ("Shortest Path Faster Algorithm") [20] は、Bellman-Ford アルゴリズムの亜種ですが、多くの場合より効率的です。SPFA アルゴリズムは、各ラウンドですべての辺を通過するのではなくて賢く見るべき辺を選択します。

このアルゴリズムでは、距離を更新しうる可能性のあるノードをキューとして持ちます。まず、アルゴリズムは開始ノード x をキューに追加します。処理では常にキューの最初のノードをみて $a \rightarrow b$ が距離を更新するとノード b をキューに追加します。SPFA アルゴリズムの効率はグラフの構造に依存します。

このアルゴリズムは多くの場合に効率的ですが最悪の場合の時間計算量は依然として $O(nm)$ で Bellman-Ford と同変わらない処理時間となる入力を作成可能です。

13.2 ダイクストラ法 - Dijkstra's algorithm

ダイクストラ法 - Dijkstra's algorithm ^{*2} は、ベルマン-フォードと同様に、始点ノードからグラフの全ノードまでの最短経路を求める手法です。ダイクストラ法は、より大規模なグラフの処理に使用できます。注意すべき点として、このアルゴリズムはグラフ内に負の重みのエッジが存在しないことが必須の条件です。

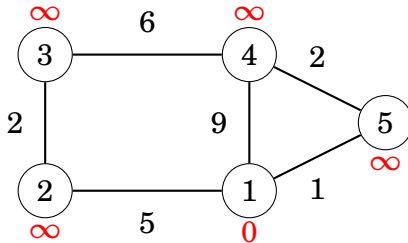
ベルマンフォードと同様に、ダイクストラ法は各ノードまでの距離を維持して、

^{*2} E. W. Dijkstra published the algorithm in 1959 [14]

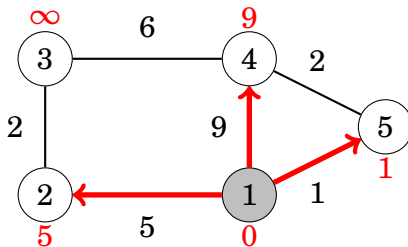
探索中に距離を更新していきます。ダイクストラ法では、グラフに負のエッジがないことを利用して、各エッジをただ一度だけ処理するので高速に動作します。

例

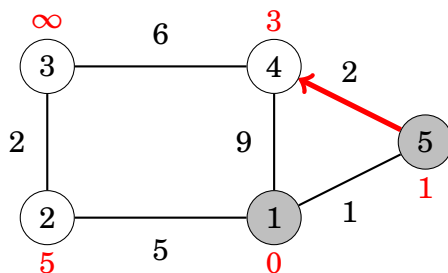
次のグラフで、ノード 1 を始点とするとき、Dijkstra のアルゴリズムがどのように働くかをみていきます。



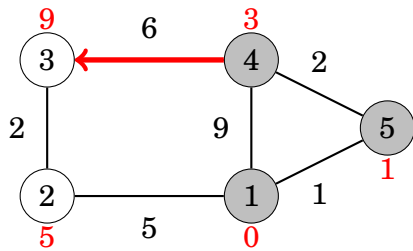
初期値はベルマン-フォードと同様に開始ノードまでの距離は 0、他のすべてのノードまでの距離は INF とします。各ステップにおいて、Dijkstra のアルゴリズムはまだ処理されていないノードの中から距離が最も小さいものを選択します。アルゴリズムが開始した際に選択される最初のノードは距離 0 のノード 1 です。ノードが選択されるとアルゴリズムはそのノードを始点とする全ての辺を調べて宛先ノードの距離を更新できるならば更新します。



この場合、ノード 1 からのエッジによって、ノード 2、4、5 の距離が更新され、その距離は 5、9、1 になりました。次の処理ノードは、距離 1 のノード 5 です。これを処理すると、ノード 4 までの距離が 9 から 3 になりました。

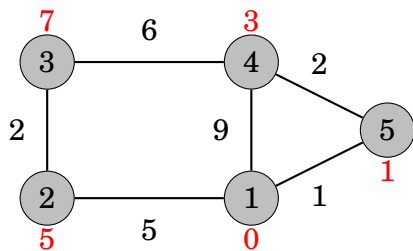


次はノード 4 であり、ノード 3 までの距離が 9 になりました。



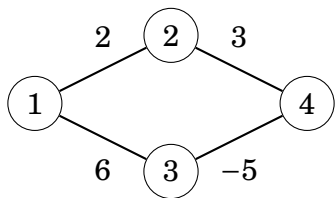
ダイクストラ法で注目する特徴は、あるノードの処理を開始するときはその距離が最終的な距離であることです。例えば、この時点では、距離 0、1、3 がノード 1、5、4 への最終的な距離である。

このように処理をすると最終的な距離は以下のようになります。



Negative edges

ダイクストラのアルゴリズムの効率性はグラフに負のエッジが含まれないことを前提にしています。もし、負の辺があるとアルゴリズムは正しくない結果を出す可能性があります。これを次のようなグラフで考えてみましょう。



ノード 1 からノード 4 への最短経路は $1 \rightarrow 3 \rightarrow 4$ で、その長さは 1 です。しかし、ダイクストラのアルゴリズムでは最小の辺を辿って $1 \rightarrow 2 \rightarrow 4$ の経路と確定します。このケースでは、重み 6 のあとに重み -5 があることが考慮されていません。

実装

グラフは隣接リストとして保存され、ノード a からノード b に重み w のエッジがあるとき $\text{adj}[a]$ には (b,w) のペアが含まれているとします。ダイクストラ法を効率的に実装するには未処理の最小距離のノードを効率的に見つけることが可能であることが最も重要です。このために距離でソートしたノードを取得可能な優先

度付きキュー (priority queue) を用います。優先度付きキューを用いると、次に処理されるノードを対数時間 (\log) で検索することができます。以下のコードでは、優先度付きキュー q は、ノード x までの現在の距離が d であることを意味する、 $(-d, x)$ という形の pair を持ちます。配列 $distance$ は各ノードまでの距離を含み、配列 $processed$ はノードが処理されたかどうかを持ちます。初期状態で x に対する距離は 0 であり、それ以外のノードに対する距離は ∞ です。

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}
```

優先度付きキューにノードへの距離をマイナスで保持しているのは C++ のデフォルトの優先度付きキューは最大要素を返すためです。最小要素を見つけるために負の距離を使用することで、デフォルトの優先度キューを直接使用することができます^{*3}。このキューには同じノードが入ることがありますが、距離が最小のものだけを処理すれば良いです。

このアルゴリズムは、すべてのノードを処理し、各エッジに対して最大で 1 つの情報を優先順位キューに追加するため、上記の実装の時間計算量は $O(n + m \log m)$ となります。

^{*3} Of course, we could also declare the priority queue as in Chapter 4.5 and use positive distances, but the implementation would be a bit longer.

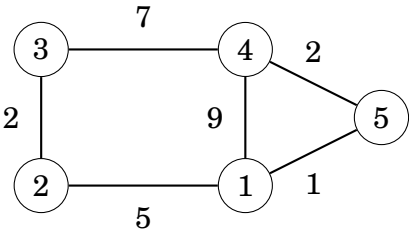
13.3 ワーシャルフロイド法 - Floyd – Warshall algorithm

ワーシャルフロイド法^{*4}は最短経路を求めるアルゴリズムですがこれまでの2つとはまた違ったアプローチです。1回の実行でグラフ上の全ての2点の最短経路を求めることができます(訳註: これまではある始点からの距離を求めていることに注意してください)。

ワーシャルフロイド法では、ノード間の距離を含む2次元の配列を保持します。まず、ノード間の直接のエッジのみを用いて距離を更新し、経由するノードを用いて距離を更新するというアプローチを取ります。

例

以下のグラフでこの動きをみていきます。



初期状態では、各ノードからそれ自身への距離は0とします。ノード a とノード b の間に重み x をの辺が存在する場合はその距離は x です。それ以外の配列の要素は INF とします。このグラフにおいて、初期配列は次のようになります。

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

このアルゴリズムは同じ操作を複数のラウンド実行します。各ラウンドにおいて、アルゴリズムは、パスの新しい中間ノードを選択し、このノードを用いて距離を減少させます。

第1ラウンドでは、ノード1が新しい中間ノードとします。ノード2とノード4の間には、ノード1が接続しているため、長さ14の新しいパスがあります。ノー

^{*4} The algorithm is named after R. W. Floyd and S. Warshall who published it independently in 1962 [23, 70].

ド 2 とノード 5 の間にも、長さ 6 の新しいパスがあります。

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	7	∞
4	9	14	7	0	2
5	1	6	∞	2	0

2 ラウンド目では、ノード 2 が新たな中間ノードとします。これにより、ノード 1 とノード 3 の間、ノード 3 とノード 5 の間に新しいパスができます。

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

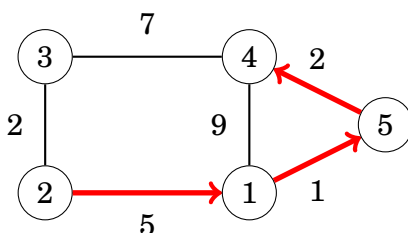
3 ラウンド目では、ノード 3 が新しい中間ラウンドとします。ノード 2 と 4 の間に新しいパスができます。

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

このアルゴリズムは、すべてのノードが中間ノードに任命されるまで、このように続けられます (訳註: 上記の例のように i ラウンド目にはノード i が中間ノードとして計算します)。アルゴリズムが終了すると、配列には任意の 2 つのノード間の最小距離が格納されています。

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

この結果からノード 2 とノード 4 の間の最短距離は 8 であることがわかります。
これを次に示します。



実装

このアルゴリズムの最も大きな利点は実装が簡単なことです。以下のコードは、`distance[a][b]` をノード a と b 間の最短距離とする距離行列とします。まず、この実装にはグラフの隣接行列 `adj` で辺の情報を与えます。

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) distance[i][j] = 0;
        else if (adj[i][j]) distance[i][j] = adj[i][j];
        else distance[i][j] = INF;
    }
}
```

この後、以下のようにして最短距離を求めます。

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            distance[i][j] = min(distance[i][j],
                                   distance[i][k]+distance[k][j]);
        }
    }
}
```

このアルゴリズムはグラフのノードを通過する 3 つの入れ子ループを含むため、時間計算量は $O(n^3)$ です。

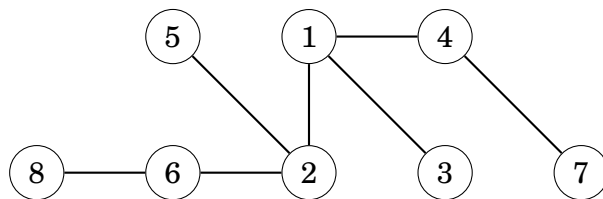
この実装は単純なので、グラフの最短経路を 1 つだけ見つける必要がある場合でも、このアルゴリズムは使用できます。ただし、このアルゴリズムが使えるのはグラフが非常に小さく 3 乗の時間計算量で十分高速に動作する場合のみです。

第 14 章

木のアлゴリズム - Tree algorithms

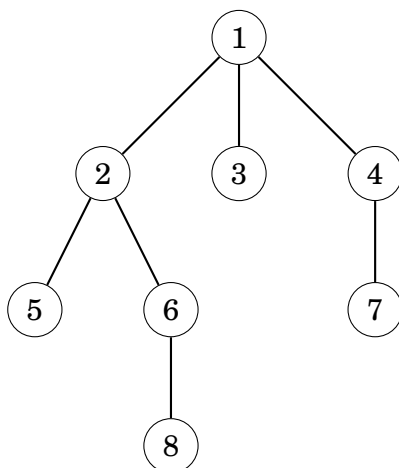
木は、 n 個のノードと $n-1$ 個のエッジからなる連結かつ閉路を含まないグラフです。木から辺を 1 つ取り除くと 2 つの要素に分かれ、辺を加えると 1 つの閉路ができます。また、木の任意の 2 つのノード間には常に一本のパスが存在します。

例えば、8 個のノードと 7 個のエッジから構成されている木の例を示します。



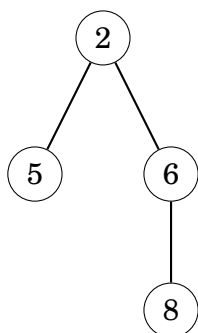
木の葉(リーフ)は、次数が 1 のノード、つまり、隣接ノードが 1 つしかないノードです。上図の葉は、ノード 3, 5, 7, 8 です。

根付き木は、あるノードを木の根として指名し、他のすべてのノードは根の下に配置されます。例えば、以下の木では、ノード 1 が根となるノード(ルートノード)です。



根付き木では、次の様に言葉を定義します。**子**は配下にいる隣接ノード、**親**は上位の隣接ノードです。根以外のノードは、ただ 1 つの親を持ちます。上図の木では、ノード 2 の子はノード 5 とノード 6 です。ノード 2 の親はノード 1 です。

根付き木の構造は**再帰的**です。あるノードに注目した時にそのノードは、そのノード自身とそのノードの子の部分木に含まれるすべてのノードを含む部分木のルートとして機能します。例えば、上図の木では、ノード 2 の部分木は、ノード 2、5、6、8 です。



14.1 木の探索 - Graph Traversal

一般的なグラフの探索アルゴリズムは、木のノードを探索に使用することができます。木には閉路がなく、親と子の関係で構成されているため木の探索は一般のグラフの探索よりも実装が簡単です。最もシンプルな方法はあるノードから深さ優先探索します。次のような再帰的な関数が考えられます。

```
void dfs(int s, int e) {  
    // process node s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

```
}

```

この関数は現在のノード s と直前のノード e が与えられます。 e はこれまでの訪問ノードの情報を持ちます。この関数は次のように呼ぶとノード x から探索を開始します。

```
dfs(x, 0);

```

$e = 0$ とすることで (訳註: この例ではノード 0 は存在しないので終わることはなく) 最初のノードは全てのノードに移動できます。

動的計画法 - Dynamic programming

動的計画法は、木の探索中に何らかの情報を計算するために用いることができます。根付き木において各ノードの部分木のノード数やそのノードから葉までの最長経路の長さといったものを $O(n)$ 時間で計算できます。

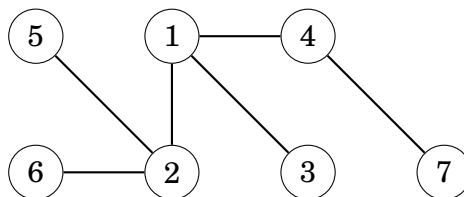
各ノード s について、その部分木に含まれるノードの数 $\text{count}[s]$ 求めてみましょう。部分木には、ノード自身とその子の部分木に含まれるすべてのノードが含まれるので以下のコードで再帰的にノード数を計算することができます。

```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}

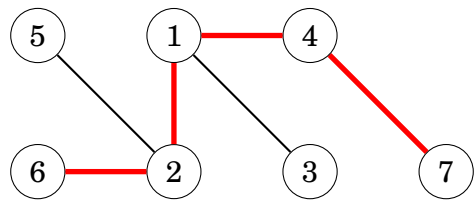
```

14.2 直径 - Diameter

木の**直径**とはその木に含まれる 2 ノード間の最長の距離です。例を示します。



この木の直径は次のパスの通り 4 です。

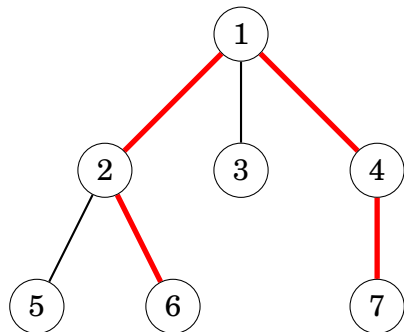


半径の経路は複数存在する可能性があります。例えば上記の例ならノード 6 をノード 5 に置き換えると、長さ 4 の別のパスを得ることができます。

木の直径を計算するための 2 つの $O(n)$ のアルゴリズムを紹介します。最初のアルゴリズムは動的計画法に基づいており、2 番目のアルゴリズムは 2 つの深さ優先探索を用います。

アルゴリズム 1: 動的計画法

木の問題を解く際によく使われるアプローチは木を根付き木と捉えることです。この木を根付き木と捉える方法は部分木について個別に問題を解けば良いケースが多々あり、有効です。このアルゴリズムはこのアプローチをとり、根付き木の各ノードにはそのパスを経由する最も長いパスが存在します。各ノードについてそのノードに含まれる最長経路の長さを表せれば、そのパスの 1 つが木の直径に相当します。例えば、以下の木では、ノード 1 が直径に相当する部分木のノードです。



各ノード x に対して以下の 2 つを求めます。

- $\text{toLeaf}(x)$: x から葉まで最大長
- $\text{maxLength}(x)$: 最高点が x であるパスの最大長

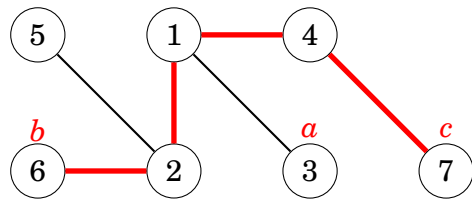
例えば上の図の木では $1 \rightarrow 2 \rightarrow 6$ があるため $\text{toLeaf}(1)=2$ となり、 $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ があるので $\text{maxLength}(1)=4$ となります。この例では $\text{maxLength}(1)$ が直径となります。

動的計画法を用いて上記の値を全ノードについて $O(n)$ 時間で計算することができます。まず、 $\text{toLeaf}(x)$ を計算するために、 x の子ノードを調べ、 $\text{toLeaf}(c)$ が最大である子 c を選んで、この値に 1 を足します。次に、 $\text{maxLength}(x)$ を計算するた

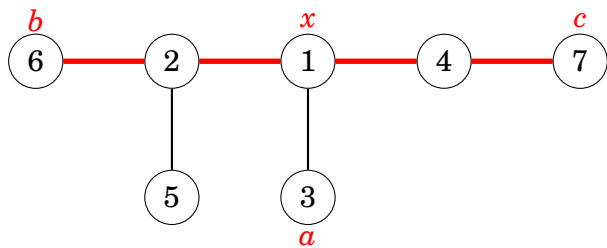
めに、 $\text{toLeaf}(a) + \text{toLeaf}(b)$ の和が最大となるような異なる二つの子 a と b を選び、この和に 2 を足します。(訳註：このような計算は木 DP と呼ばれます)

アルゴリズム 2: 深さ優先探索

木の直径を計算するためにもう一つの効率的な方法を紹介します。2 回の深さ優先探索を利用した方法です。まず、木の任意のノード a を選び、 a から最も遠いノード b を見つけます。 b から最も遠いノード c を見つけます。木の直径は b と c の間の距離となります。次のグラフで a, b, c を次のように置くことができます。



たったこれだけです。本当にうまく動作するのでしょうか？ ツリーの向きを変えて直径のパスが水平になるように表現します。

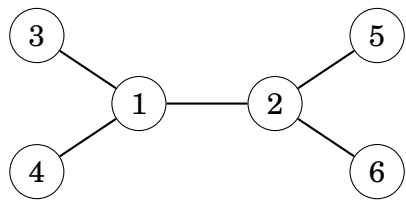


ノード x は、ノード a からの経路が直径に対応する経路に合流するノードとしましょう。 a から最も遠いノードは b とはノード c あるいはノード x から少なくとも同じだけ遠い異なるノードです。したがって、このノードは直径に対応するパスの終点として常に有効な候補となります。(訳註: a から辿るとこの表現をした際の水平の線に必ず合流します。なので 1 回目の BFS でそこから最も遠い距離を探すことはいずれかの水平な線の候補の端点を見つけることとなり、そこから BFS を再度行えば同じ様にもう一方の端点を探せます)

14.3 全ノードからの最長経路 - All longest paths

木の各ノードについて、そのノードから始まる最大の経路の長さを計算することを考えましょう。これは木の直径問題の一般化とも言えます。それらの長さのうち最大のものは木の直径だからです。

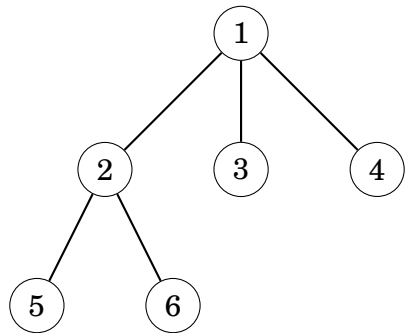
この問題は $O(n)$ で解くことができます。



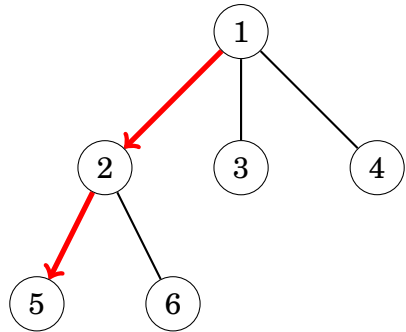
$\text{maxLength}(x)$ を x を始点とするパスの最大長とします。上図の木では $\rightarrow 1 \rightarrow 2 \rightarrow 6$ というパスがあるので、 $\text{maxLength}(4) = 3$ です。これを表にします。

ノード x	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

先ほどと同じ様に根付き木にすると見通しがよくなります。

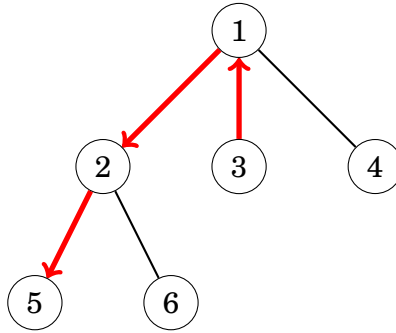


各 x について、その子のノードの中で最長のパスを持つノードを探します。例えば、ノード 1 からの最長の経路はその子 2 を通ることがわかります。

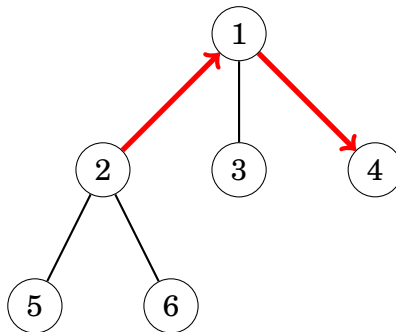


これは前回と同様に動的計画法で $O(n)$ で解くことができます。例えば、ノード 3 からの最長経路はその親 1 を通る。

次に、各ノードに対して、以下の計算を行います。各ノード x について、親 p を通る経路の最大長を計算します。例えば、ノード 3 からの最長経路はその親 1 を通ります。



これは p からの最長経路を選べばよさそうにみえますが、 p からの最長経路が x を経由する場合があるため、必ずしもうまくいかきません。次の様な例があります。



ですが、各ノード x について 2 つの最大長を記憶することで、 $O(n)$ で解くことができます。まずは次の 2 つを定義しましょう。

- $\text{maxLength}_1(x)$: x からの最大のパス長
- $\text{maxLength}_2(x)$ それとは異なる次の最大のパス長。長さは同じこともある。

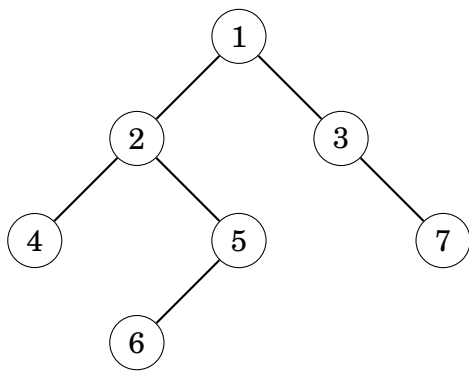
上記のグラフでは $1 \rightarrow 2 \rightarrow 5$ から $\text{maxLength}_1(1) = 2$ であり、 $1 \rightarrow 3$ から $\text{maxLength}_2(1) = 1$ となります。

$\text{maxLength}_1(p)$ に対応するパスが x を通る時、 $\text{maxLength}_2(p) + 1$ ということができます。そうでない場合は、 $\text{maxLength}_1(p) + 1$ となります。

14.4 二分木 - Binary trees

二分木とは、全てのノードが左と右に部分木を持つ根付きの木のことである。ただし、部分木が空であることも許容します。言い換えれば、二分木のすべてのノードは 0 個、1 個、または 2 個の子を持つと言えます。

例を示します。



二分木のノードには 3 つの自然な走査 (Traversal) があります。

- **pre-order:** ルートを処理してから左を処理する。次に右を処理する。
- **in-order:** 左を処理してからルートを処理する。次に右を処理する。
- **post-order:** 左を処理してから右を処理する。次にルートを処理する。

上記の木では、pre-order は [1,2,4,5,6,3,7], in-order は [4,2,6,5,1,3,7], post-order は [4,6,5,2,7,3,1] となります。

また、pre-order と in-order の 2 つが分かると木が復元できることも知られています。例えば、pre-order[1,2,4,5,6,3,7] で in-order[4,2,6,5,1,3,7] とわかれば上の木を復元できます。同様に post-order と in-order がわかっている場合も復元できます。

ただし、pre-order と post-order だけがわかっている場合は注意が必要です。条件を満たす複数の可能性があるからです。この例をみてみます。



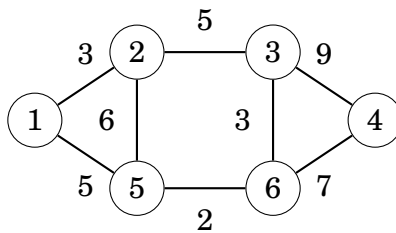
どちらの木も pre-order [1,2] で post-order is [2,1] です。このため、どちらかに確定させることはできません。

第 15 章

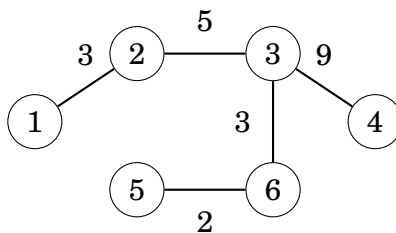
全域木 - Spanning trees

全域木 (spanning tree) は任意の 2 つのノード間にパスが存在するようなエッジで構成されている木です。一般的な木と同様、木は連結していて閉路は存在しません。木を構成するにはいくつかの方法があります。

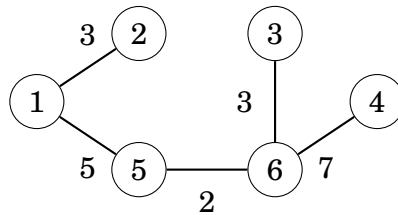
次のようなグラフを考えてみましょう。



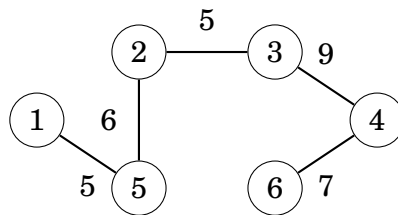
例えば、1 つの全域木は次のようになります。



全域木の重さとはエッジの重みの合計です。上の全域木の重みは、 $3+5+9+3+2=22$ です。**最小全域木 (minimum spanning tree)** は、全ての全域木の中で最も重さが小さな全域木です。先ほどの例では、次の最小全域木は重さが 20 です。



最大全域木 (maximum spanning tree) は重さが最大となる木です。先の例では次のように重さ 32 の最大全域木ができます。



ここで、最小全域木・最大全域木になるグラフ（木）は複数存在することがあります。これらの全域木は貪欲な方法で求めることができます。この章では、2つのアルゴリズムについて説明します。説明では最小全域木のように絞って説明を行います。最大全域木の場合は処理する辺の順を逆とすれば良いです。

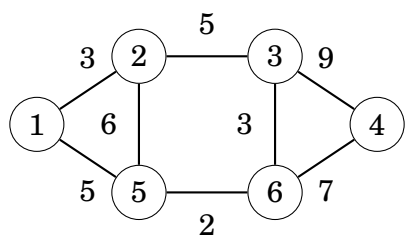
15.1 クラスカル法 - Kruskal's algorithm

Kruskal 法^{*1} は次のように動作します。まず、辺がなくノードだけのグラフを考えます。次に、このアルゴリズムは重みが小さい順に辺を調べて閉路を作らない場合は常にグラフにエッジを追加します。このアルゴリズムは木を連結成分を保持しながら行います。初期状態は各ノードは別々の連結成分に属しており、木にエッジを追加する時にその 2 つの成分を結合します。最終的にすべてのノードは同じ成分に属し、最小全域木となります。

^{*1} このアルゴリズムは 1956 年に J. B. Kruskal によって発表された [48].

例

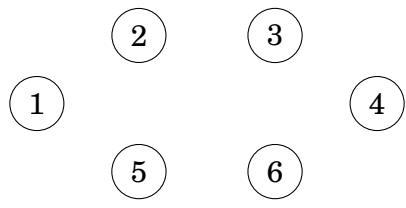
次のグラフを考えます。



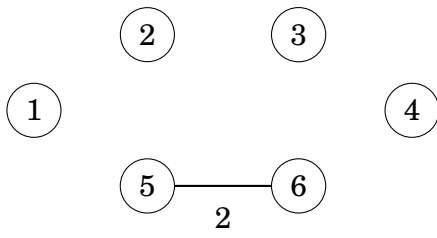
最初に全てのエッジを重みが小さい順にソートします。

edge	weight
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

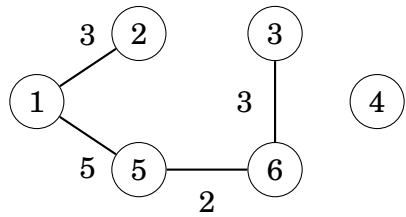
リストを上から処理し、各辺が2つの別の連結成分を結合するなら、その辺を接続します。



まず、5-6 の処理は {5} と {6} を連結し、{5,6} とします。



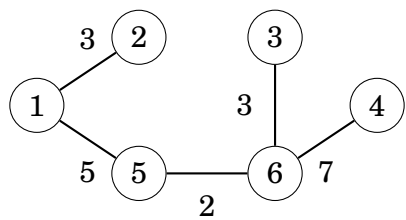
同じように、1-2, 3-6, 1-5 が接続されます。



これらのステップの後、ほとんどの連結成分は結合され、2つの連結成分が存在します。 $\{1,2,3,5,6\}$ と $\{4\}$ です。

次に処理するのは 2-3 ですが、同じ連結成分に含まれるので接続しません。2-5 も同様です。

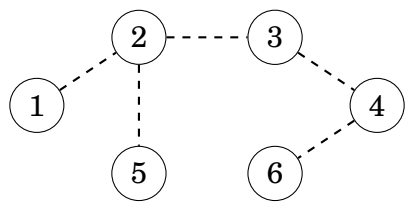
最後に、4-6 が接続されます。



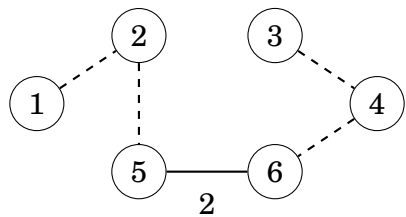
これで全ての点が同じ連結成分に含まれるので処理を終えます。重さが $2+3+3+5+7=20$ の最小全域木ができました。

なぜこれで良いか？

これでなぜ最小全域木が求められるのでしょうか？ 最小の重みの辺が含まれないことを考えます。例えば、先ほどの例で最小重みの辺 5-6 が含まれないとします。そのようなグラフは複数考えられますが、例えば次のような木を考えましょう。



しかし、この木が最小全域木ではありません。木からある辺を取り除き、5-6 に置き換えることができ、さらに重さの小さな木ができます。



このため、最小の重さの辺を最小全域木に含めることは妥当で最小のスパニングツリーが生成されます。重さが小さな順に辺を木に加えることが最適であることを示すことができ、クラスカルのアルゴリズムは正しく動作して常に最小全域木となることが示されました。

実装

クラスカル法の実装は辺をリスト表現で持つのが便利です。最初に $O(m \log m)$ でリスト中のエッジを重さ順にソートし、次に以下のように最小スパニングツリーを構築します。

```
for (...) {
    if (!same(a,b)) unite(a,b);
}
```

ループは a, b を処理します。 $same$ は同じ連結成分にいるかを判定する関数で、 $unite$ は互いに所属する連結成分を結合する操作です。これらをシンプルに書くと $O(\log(n+m))$ となってしまいますが $Union-Find$ を用いるとこれらは $O(\log n)$ で実装することができるため、ソート後の計算量 $O(m \log n)$ を実現できます。

15.2 Union-find 構造 - Union-find structure

Union-find 構造 は集合の集まりを管理するデータ構造です。どの要素もいずれかの集合に属し、2 つ以上の集合に属しません。この構造は先にあげた $unite$ と $same$ を $O(\log n)$ で実行できます。^{*2}

構造

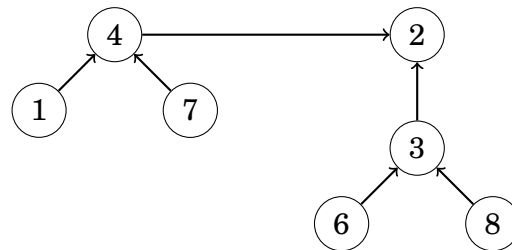
Union-find では各集合の 1 つの要素がその集合の代表として存在します。そして、集合の他の要素は必ず代表への連鎖に辿ることができます。

集合が $\{1, 4, 7\}$ 、 $\{5\}$ 、 $\{2, 3, 6, 8\}$ としましょう。

^{*2} ここで紹介する構造は、1971 年に J. D. Hopcroft と J. D. Ullman によって紹介されたもの [38]。その後、1975 年に R. E. Tarjan がこの構造のより洗練された変種を研究し [64]、現在では多くのアルゴリズムの教科書で議論されています。



この図では集合の代表は 4、5、2 です。あるの要素の代表は、その要素から木を辿ると発見できます。例えば、 $6 \rightarrow 3 \rightarrow 2$ という連鎖をたどれば、要素 2 が要素 6 の代表です。2 つの要素が同じ集合に属するのは、その代表が同じであるときだけです。2 つの集合は、一方の集合の代表と他方の集合の代表を結ぶことで結合することができます。例えば、 $\{1, 4, 7\}$ と $\{2, 3, 6, 8\}$ は以下のように結合することができます。



結果、 $\{1, 2, 3, 4, 6, 7, 8\}$ の要素が集合に含まれ、要素 2 がそれらの集合全体の代表となり、もともと代表要素 4 であった要素 2 を指すようになります。Union-find の操作の計算量は集合をどのように結合するか依存します。良い工夫は小さい集合を大きい集合の代表要素に繋がります (同じ大きさの集合の場合どちらを大きい方とみなしてもよいです)。この方法を用いるとどのパスの長さも $O(\log n)$ となるので、対応する辺を辿る中で任意の要素の代表を効率的に求めることができます。

実装 - Implementation

union-find は、配列を使用して実装できます。以下の実装では、配列 link は各要素について、親となる要素 (代表であればその要素自身) を含み、配列 size は各代表について、対応する集合の大きさを保持します。

初期状態では、各要素は別々の集合に属しています。

```

for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
  
```

関数 find は、ある要素 x の代表を返します。代表は x から始まる鎖をたどって見つけることができます。

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

関数 same は要素 a, b が同じ集合に属するかどうかをチェックする関数で、関数 find を使うことで簡単に実現できます。

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

関数 unite は、要素 a と b を含む集合を結合する関数です。この時、要素は異なる集合に存在しなければなりません。この関数は、まず集合の代表を求め、次に小さい方の集合を大きい方の集合に接続します。

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

関数 find の時間計算量は、各鎖の長さが $O(\log n)$ であると仮定すると、 $O(\log n)$ で動作します。この場合、関数 same と unite も $O(\log n)$ の時間で動作します。関数 unite は、小さい集合を大きい集合に接続することで、各木の長さが最長で $O(\log n)$ であることを保証できます。

15.3 プリムのアルゴリズム - Prim's algorithm

プリムのアルゴリズム - Prim's algorithm^{*3} は最小全域木を求めるための手法です。このアルゴリズムでは、まず各ノードを独立した木に追加します。その後は常に最小コストの辺を選択して新しいノードを木に追加します。最後に、すべてのノードが一つの木に追加され最小全域木が発見されます。

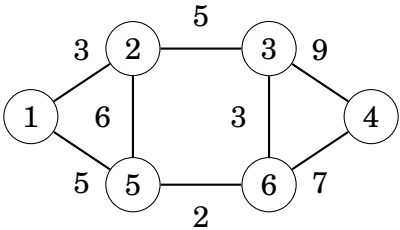
プリムのアルゴリズムはダイクストラのアルゴリズムに似ています。ダイクストラのアルゴリズムは常に開始ノードからの距離が最小となるエッジを選択します

^{*3} The algorithm is named after R. C. Prim who published it in 1957 [54]. However, the same algorithm was discovered already in 1930 by V. Jarník.

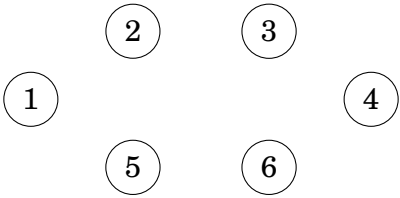
が、プリムのアルゴリズムは単に新しいノードを木に追加する最小コストの辺を選んでいきます。

例 - Example

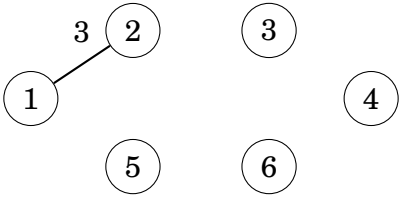
動作を図を見ていきましょう。



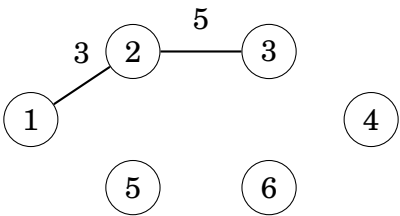
最初は全てのノードは独立しています。



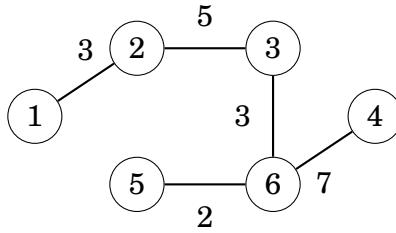
任意のノードを開始ノードとしてよいのでノード 1 を選択したとします。まず、重み 3 のエッジで結ばれているノード 2 を追加しましょう。



この後、重み 5 を持つ辺は 2 本あるので、ノード 3 かノード 5 のどちらかを木に追加できます。ノード 3 を追加します。



これを繰り返します。



実装 - Implementation

ダイクストラのアルゴリズムと同様に、プリムのアルゴリズムも優先度付きキューを用いて効率的に実装できます。優先度付きキューには、1本のエッジを使って現在のコンポーネントに接続できるすべてのノードとそのエッジの重みの昇順に格納しておきます。

プリムのアルゴリズムの時間計算量は $O(n + m \log m)$ で、ダイクストラのアルゴリズムの時間計算量と同じです。実際には、プリムのアルゴリズムとクラシカルなアルゴリズムはどちらも効率的で、アルゴリズムの選択は好みの問題です。ただし、クラシカル法が用いられることが多いです。

第 16 章

有向グラフ - Directed graphs

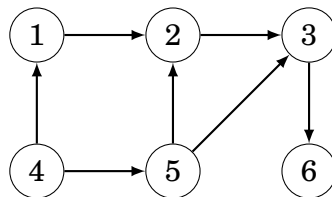
ここでは 2 つの特徴を持つグラフについて検討していきます。

- **非巡回グラフ - Acyclic graphs:** 閉路を含まない有向グラフのこと。時に DAG(Directed Acyclic Graphs) と呼ばれる
- **Successor graphs:** 各ノードの出次が 1 であるグラフ。このため、ただ一つの子を持つ。

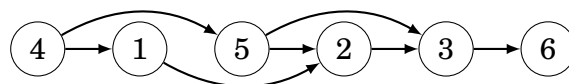
どちらの場合も、グラフの特別な性質に基づく効率的なアルゴリズムがあります。

16.1 トポロジカルソート - Topological sorting

トポロジカルソート - **topological sort** とは、有向グラフのノードにおいて、ノード a からノード b へのパスが存在する場合、ノード a がノード b の前に現れるような順序を持つグラフのことです。



このトポロジカルソートは [4,1,5,2,3,6] です。



非巡回グラフは常にトポロジカルソートです。しかし、グラフが閉路を含む場合は閉路のどのノードも順序において閉路の他のノードより前に現れることができな

いのでトポロジカルソートを形成することはできません。深さ優先探索を用いて、有向グラフに閉路があるかどうかを調べ、閉路がない場合にはトポロジカルソートを構成できると判定できます。

アルゴリズム - Algorithm

ベースとなる考え方はグラフのノードを適当にみていき、まだ処理されていない場合はそのノードから深さ優先探索を開始していきます。ノードは3つのいずれかの状態を持ちます。

- 状態 0: ノードは処理されていない状態 (白色)
- 状態 1: ノードが処理中 (薄いグレー)
- 状態 2: ノードが処理された状態 (濃いグレー)

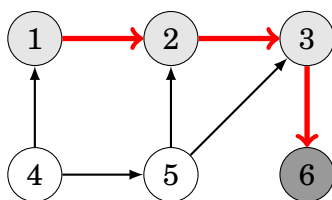
初期状態では、各ノードの状態は0です。初めて検索が到達するとノードの状態は1になります。そして、そのノードのすべての探索が処理された後、そのノードの状態は2にします。

グラフに閉路が存在する場合、状態が1であるノードに探索をするので探索中にグラフに閉路があることが判明します。

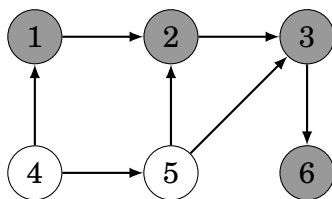
グラフが閉路を含まない場合はこの条件以外の時で、全てのノードの状態が2になったときに各ノードの探索順をリストとすることで、トポロジカルソートとなります。

Example 1

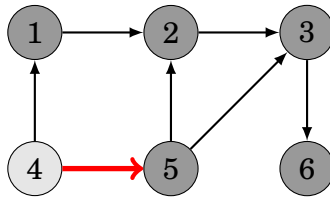
このグラフの例では1,2,3,6と探索が進みます。



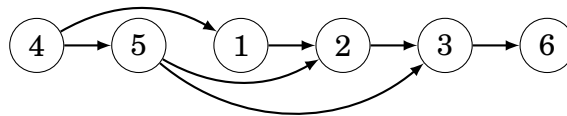
ここで、6には探索する先がないので、状態2となり、リストに値が登録されます。次に、3,2,1と追加されていきます。



いま、リストの状態は [6,3,2,1] です。次は 4 が探索されます。



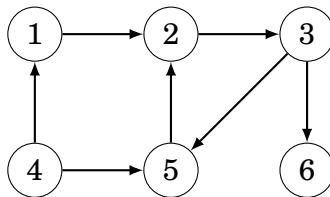
このようにリスト [6,3,2,1,5,4] を得ることができます。この逆順がトポロジカルソート [4,5,1,2,3,6] です。



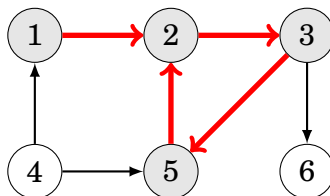
なお、トポロジカルソートは1つとは限りません。閉路を含まないグラフでは複数のトポロジカルソートが存在します。

Example 2

閉路を含むのでトポロジカルソートできない場合はどうなるでしょうか。



次のように進んでいきます。



探索は状態が 1 であるノード 2 に到達しました。つまり、グラフが閉路を含むということです。この例では、 $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$ という閉路が存在します。

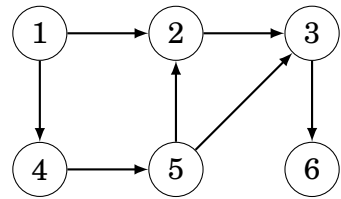
16.2 木 DP - Dynamic programming

有向グラフが閉路を含まないのであれば動的計画法を木上で適用することができます。例えば、始点ノードから終点ノードへの経路に関する次のような問題を効率的に解くことができます。

- パスの種類は？
- 最短あるいは最大のパスは？
- 最短あるいは最大の辺の数は？
- 全てのパスにおいて必ず通るノードはあるか？

パス数のカウント

次のグラフで 1 から 6 のパスの数を考えます。



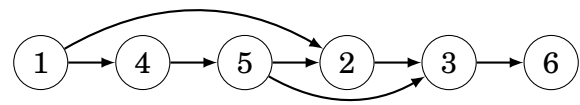
このようなパスは以下の 3 つがあります。

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

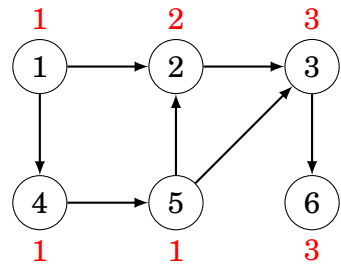
$\text{paths}(1) = 1$ です。 $\text{paths}(x)$ は以下のように再帰的に求めています。

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \cdots + \text{paths}(a_k)$$

は x への辺が存在するノードです。グラフは閉路を持たないため、 $\text{path}(x)$ の値はトポロジカルソートの順に計算することができます。上のグラフのトポロジカルソートの 1 つは以下の通りです。



このためパス数は以下のように求められます。

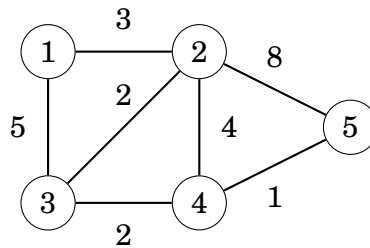


$\text{paths}(3)$ を求める時、ノード 2 と 5 からノード 3 への辺があるので、 $\text{paths}(2) + \text{paths}(5)$ を計算します。 $\text{paths}(2) = 2$, $\text{paths}(5) = 1$ ですから、 $\text{paths}(3) = 3$ となり

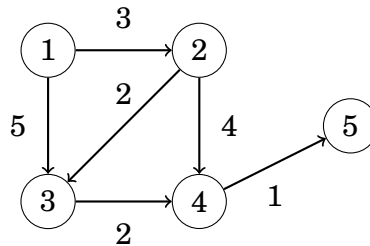
ます。

ダイクストラのアルゴリズムの拡張

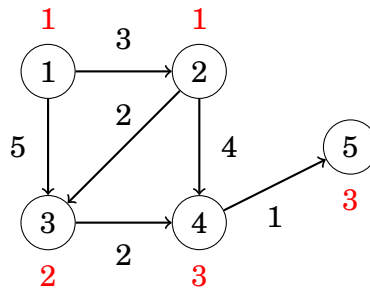
ダイクストラのアルゴリズムを利用すると、開始ノードから各ノードへの最短経路を使用してそのノードに到達する可能性を示す有向の閉路が存在しないグラフを得ることができます。このグラフに対しては動的計画法を適用することができます。以下を考えます。



ノード 1 からの最短経路は以下のようになります。



これでノード 1 から 5 までの最短経路を以下のように求めることができます。

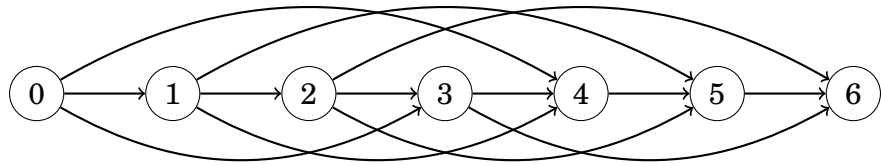


グラフによる問題表現 - Representing problems as graphs

動的計画問題は閉路が存在しない有向のグラフで表現することができます。このようなグラフでは、各ノードが動的計画法の状態に対応し、辺は状態各ノードの依存を示します。

例として、硬貨 $\{c_1, c_2, \dots, c_k\}$ を用いて金額 n を作る問題を考えます。この問題では、各ノードが金額に対応し、エッジがコインの選び方を示すグラフと捉えるこ

とができます。たとえば、コイン $\{1,3,4\}$ と $n=6$ の場合、グラフは次のようになります。



この表し方をするとノード 0 からノード n までの最短経路はコインの枚数が最小の解に対応していることとなり、ノード 0 からノード n までの経路の総数は解の総数に等しくなります。

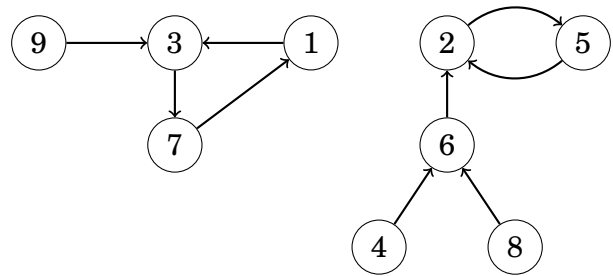
16.3 サクセスパス - Successor paths

successor graphs に焦点を当てていきます。このグラフでは、各ノードの出次辺は 1、つまり、ちょうど 1 本の辺が各ノードから出ています。サクセスグラフは 1 つ以上のコンポーネントとからなます。それぞれのコンポーネントには 1 つの閉路とそれにつながるパスがあります。サクセスグラフは **functional** グラフと呼ばれることもあります。これはサクセスグラフがグラフの辺を定義する関数に対応するからです。関数のパラメータはグラフのノードであり、関数はそのノードの後ろの後ろ、と考えます。

例えば以下の関数を考えます。

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

これは次のようなパスと考えられます。



successor グラフの各ノードは一意的な次のノードを持つので、ノード x から始めて k 歩進んで到達するノードを与える関数 $\text{succ}(x, k)$ を定義することができます。たとえば、上のグラフでは $\text{succ}(4, 6) = 2$ で、これはノード 4 から 6 回移動するとノード 2 に到達することを表します。



$\text{succ}(x, k)$ 値を計算する簡単な方法は、ノード x から始めて k 歩を実際に進むことですが、これには $O(k)$ 個の時間がかかってしまいます。前処理を行うことと $O(\log k)$ で計算することができます。

この方法を紹介します。 k が 2 の累乗であって、最大でも u である $\text{succ}(x, k)$ のすべての値を事前に計算します。 u は必要となる最大の数です。これは以下のような再帰的な方法で求めることができます。

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

各ノードに対して $O(\log u)$ 個の値を計算するため、値の事前計算には $O(n \log u)$ 個の時間がかかります。

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x, 1)$	3	5	7	6	2	2	1	6	3
$\text{succ}(x, 2)$	7	2	1	2	5	5	3	2	7
$\text{succ}(x, 4)$	3	2	7	2	5	5	1	2	3
$\text{succ}(x, 8)$	7	2	1	2	5	5	3	2	7
...									

この事前計算を用いるとステップ数 k を 2 の累乗の和で表すことで、 $\text{succ}(x, k)$ の任意の値を算出することができます。例えば、 $\text{succ}(x, 11)$ の値を計算したい場合は、 $11 = 8 + 2 + 1$ とします。こうすることで、

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

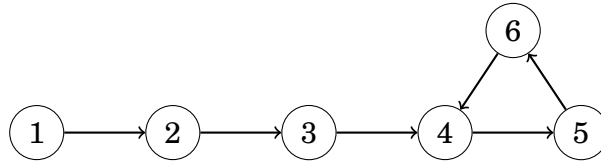
これは次のように展開されます。

$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

これによって全ての数は $O(\log k)$ で求めることができます。(訳註: ダブリングと呼ばれるテクニックです)

16.4 閉路検出 - Cycle detection

閉路で終わる有向グラフを考えます。始点から歩き始めたときに閉路の最初のノードは何で閉路はいくつのノードを含むか？ に焦点をあてます。



ノード 1 から歩き始めたときに閉路に属する最初のノードは 4 で、この閉路は 3 つのノード (4、5、6) から構成されています。閉路を検出する簡単な方法は、グラフ内を順に探索していき訪問済みかを記録しておくことです。あるノードに 2 度目に達した時にそのノードは閉路に属するの最初のノードです。この方法は $O(n)$ で動作します。

ここで閉路検出のための空間計算量が $O(1)$ であるフロイドの循環検出アルゴリズムを紹介します。

フロイドの循環検出アルゴリズム - Floyd's algorithm

フロイドの循環検出アルゴリズム - **Floyd's algorithm** とは^{*1} 2 つのポインタ a 、 b を用いてグラフを探索していく方法です。両ポインタはグラフの始点であるノード x に位置します。そして、1 ターンごとに、ポインタ a は 1 歩、ポインタ b は 2 歩前進させていきます。これを両方のポインタが (開始時点を除いて) 最初と同じ位置に重なるまで続けます。

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

k 歩で出会った時、ポインタ a は k 歩、ポインタ b は $2k$ 歩歩いているので、閉路の長さは k で割ることができます。このため、 a をノード x に移動して再び a と b が重なるところまでポインタを進めると閉路に所属する最初のノードを知ることができます。

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
```

^{*1} The idea of the algorithm is mentioned in [46] and attributed to R. W. Floyd; however, it is not known if Floyd actually discovered the algorithm.

```
first = a;
```

そして、閉路の長さは以下のように求めることができます。

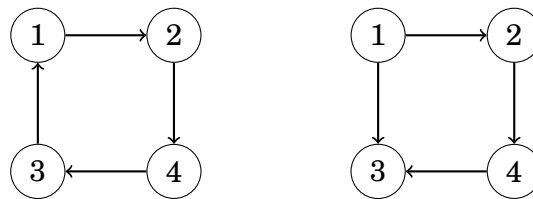
```
b = succ(a);  
length = 1;  
while (a != b) {  
    b = succ(b);  
    length++;  
}
```


第 17 章

強連結 - Strong connectivity

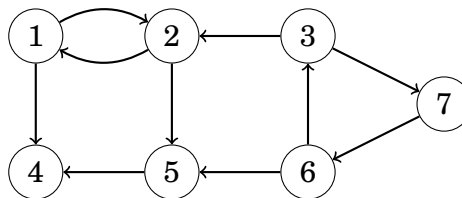
有向グラフは辺は一方方向にしか進めないののために、あるグラフ上のあるノードから別のノードへの経路が存在することは保証されません。そこでノードとノードに辺があるとは違う新しい概念の定義に意味があります。

グラフ内の任意のノードから他のすべてのノードへのパスが存在する場合、グラフは**強連結 - strongly connected**であるといいます。次の図では左のグラフは強連結ですが右のグラフは強連結ではありません。

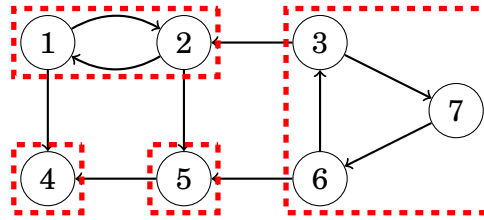


具体的には、右のグラフは 2 から 1 へ到達できないために強連結グラフといえません。

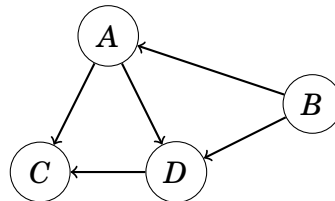
強連結成分 - strongly connected components は、元のグラフを閉路を含まないようにできるだけ大きくする成分グラフから形成したグラフです。



強連結成分は以下のようになります。



これに対応する成分グラフは次の通りとなります。



ここでの成分は $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$, $D = \{5\}$ となります。

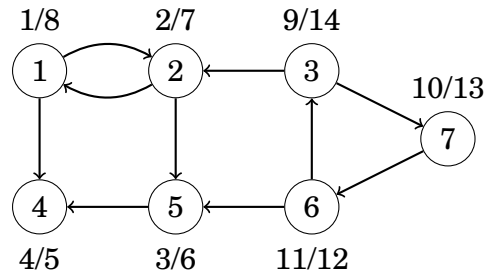
この成分グラフは閉路を含まない有向グラフであるため、元のグラフよりも処理しやすくなります。閉路が含まれないので確実にトポロジカルソートの操作ができ、16章で紹介したような動的計画法を利用することができます。

17.1 Kosaraju's algorithm

Kosaraju's algorithm^{*1} は有向グラフの強連結成分を効率的に求める方法です。このアルゴリズムでは、2回の深さ優先探索を行います。1回目の探索で探索でグラフの構造に従ってノードのリストを構築し、2回目の探索で強連結成分を求めます。

1 回目の探索 - Search 1

まず、深さ優先探索が処理する順番にノードのリストを構築します。未処理の各ノードで深さ優先探索をし、リストに追加します。このグラフの例では、以下の順序でノードが処理されていきます。



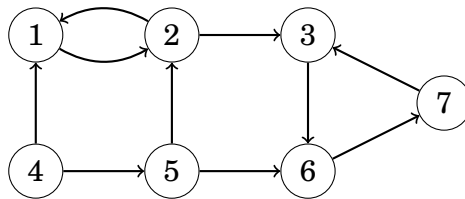
^{*1} According to [1], S. R. Kosaraju invented this algorithm in 1978 but did not publish it. In 1981, the same algorithm was rediscovered and published by M. Sharir [57].

x/y という表記は x に探索が始まり、 y に探索が終わったことを示します。

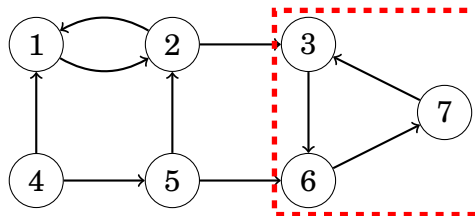
ノード番号	処理完了時間
4	5
5	6
2	7
1	8
6	12
7	13
3	14

2 回目の探索 - Search 2

次に強連結成分を求めていきます。まず、グラフのすべてのエッジを逆向きに反転させます。これによって、余分なノードを持たない強連結成分を必ず見つけることが保証されます。辺を反転させると次のようなグラフになります。

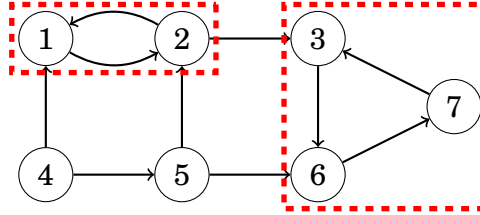


この後、最初の検索で作成されたノードのリストを逆順に捜査します。ノードが成分に属さない場合、新しい成分を作成してその探索中に見つかったすべての新しいノードを新しい成分に追加する深さ優先探索を行います。このグラフの例では、最初のコンポーネントはノード 3 から始まっています (最初の探索で最後のノードは 3 だったことを思い出してください)。

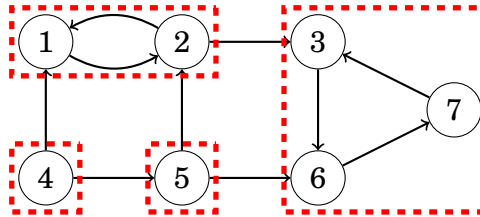


全てのグラフは反転しているため、成分が他の成分にリークすることはありません。

リストの次のノードは7と6ですが、これらは既に成分に属しているなのでこの成分の探索は完了します。次にノード1から新しい成分の探索を開始します。



同じように次の成分として5。そして4が処理されます。



これは深さ優先探索を2回行うだけなので計算量は $O(n + m)$ となります。

17.2 2-SAT 問題 - 2SAT problem

強連結は **2-SAT 問題 - 2SAT problem**^{*2}. と深く関係します。これは次のような式で与えられる問題です。

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

各 a_i と b_i はそれぞれ論理変数 (x_1, x_2, \dots, x_n) あるいは論理変数の否定 $(\neg x_1, \neg x_2, \dots, \neg x_n)$. で示されます。ここで使ったシンボルの "∧" と "∨" はそれぞれ論理演算の "and" と "or" です。

2-SAT 問題はこの解となる答えを求めるか、そのような組み合わせは存在しないと示す問題です。

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

という問題を考えると次の割り当て例は条件を満たします。

^{*2} The algorithm presented here was introduced in [4]. There is also another well-known linear-time algorithm [19] that is based on backtracking.

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

次の式は答えが存在しません。

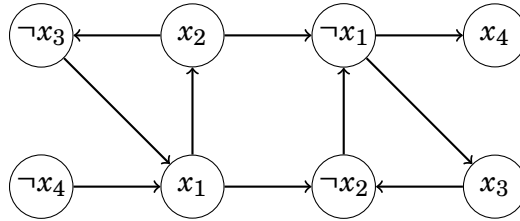
$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

これはどのような割り当てを行ったとしても x_1 に矛盾が生じてしまうのです。

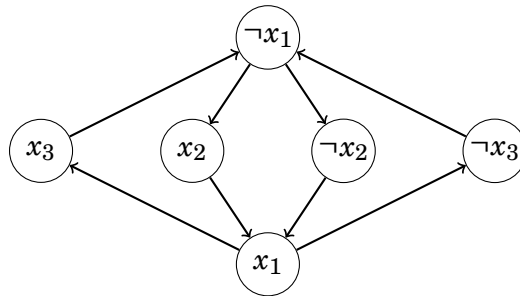
x_1 が false なら x_2 と $\neg x_2$ が true でないといけませんがこれは不可能です。 x_1 が true なら x_2 と $\neg x_2$ が false でないといけませんがこれも不可能です。

2-SAT 問題はノードが次の対応するグラフとして表現することができます。論理変数 x_i とその否定 $\neg x_i$ のノードがあり、エッジは変数間の接続だとします。それぞれの $(a_i \vee b_i)$ は次の 2 つのエッジを生成します。 $\neg a_i \rightarrow b_i$ と $\neg b_i \rightarrow a_i$ です。。これは、 a_i が成立しない場合に b_i は必ず成立し、その逆も必ず成立することを意味します。

L_1 をグラフで表現すると次のようになります。



同じように L_2 も次のように表現できます。

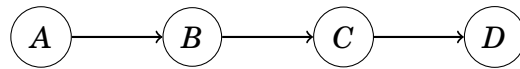


グラフを用いて式が真となるように変数の値を割り当てることが可能かどうかを調べられます。ある強連結成分に論理変数 x_i とその否定 $\neg x_i$ が同時に所属しなければ、この式を成立させる組み合わせが存在するといえます。逆に両ノードが同じ強連結成分に所属する時、その 2 つの変数は同時に満たされなければならない、成立

する組み合わせがないといえます。この条件によって式 L_1 のグラフはその解が存在することがわかり、 L_2 は成立する組み合わせがないことがわかります。

解が存在する場合は成分グラフをトポロジカルソートの逆順に見ていくことで、変数の割り当てを求めることができます。各ステップで、未処理成分に含まれる処理をしていきます。ある成分をみて値が割り当て割れていなければ、成分内の変数に値が割り当てられていない場合、その値は成分内の値に従って決定します。もし割り当てが決まっているならばなにもしません。

例えば、 L_1 のグラフのトポロジカルソート後の接続は以下のようになっています。



各成分は次の通りです。 $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$, $D = \{x_4\}$ 。

この解となる割り当てを決める時、まず x_4 が true となる成分 D を処理します。その後、構成要素 C を処理し、 x_1 と x_2 が false、 x_3 が true となります。

これで全ての論理変数に値が割り当てられたので、残りの構成要素 A と B は割り当てを変更しません。

このような割り当てができるのはグラフが特殊な構造だからです。ノード x_i からノード x_j へ、ノード x_j からノード $\neg x_j$ へのパスがある場合、ノード x_i は決して真にはなりません。この理由は、ノード $\neg x_j$ からノード $\neg x_i$ そして x_i and x_j の両方が偽になるからです。

さらに難しい問題として、式の各部分が $(a_i \vee b_i \vee c_i)$ のような形になる 3-SAT 問題があります。ただしこの問題は NP 困難であり、この問題を解く効率的なアルゴリズムは知られていない。

第 18 章

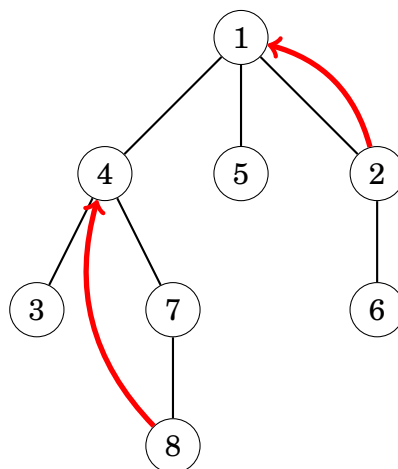
木に対するクエリ - Tree queries

ここでは根付き木における部分木とパスに関するクエリについて説明します。例えば次のようなクエリを取り上げます。

- k 番目の祖先 (ancestor) はどのノード?
- ある部分木の値の合計は?
- 2つのノード間のパス上の値の和は?
- 2つのノードの共通祖先 (LCA) は?

18.1 祖先の検索 - Finding ancestors

根付き木においてノード x の k 番目の祖先 (ancestor) とは x から k 個だけ根の方向に移動したときのノードです。ノード x の k 番目の祖先を $ancestor(x, k)$ とします。(祖先が存在しない場合は 0 とする)。次の木では、 $ancestor(2, 1) = 1$ であり $ancestor(8, 2) = 4$ です。



ancestor を求める最もシンプルな方法は k 回の移動を実際に行うことです。この方法の時間計算量は $O(k)$ であり、 n 個のノードを持つ木では n 個のノードの探索を持つ可能性があるため、高速に動作するとは言えません。

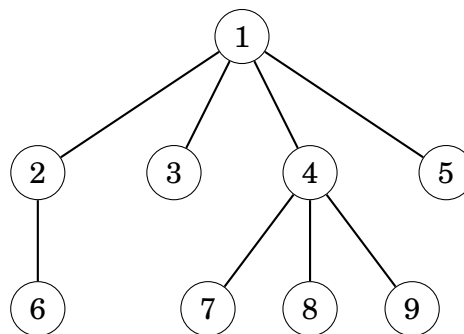
ここで 16.3 章で用いたのと同様の手法 (訳註: ダブリング) を用いれば、前処理を行うことで $ancestor(x, k)$ を $O(\log k)$ で効率的に求められます。 $k \leq n$ となる 2 の累乗について $ancestor(x, k)$ を事前計算します。上記の木に対する値は以下の通りになります。

x	1	2	3	4	5	6	7	8
$ancestor(x, 1)$	0	1	4	1	1	2	4	7
$ancestor(x, 2)$	0	0	1	0	0	1	1	4
$ancestor(x, 4)$	0	0	0	0	0	0	0	0
...								

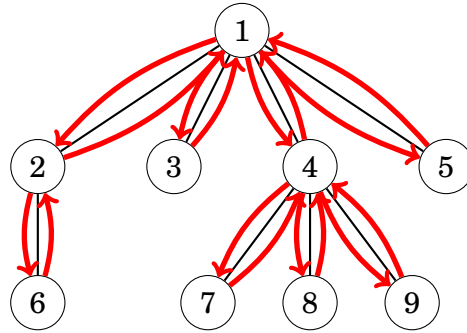
この前処理には各ノードに対して $O(\log n)$ の計算が必要なので $O(n \log n)$ の時間がかかります。 $ancestor(x, k)$ の任意の値は k を各項が 2 のべき乗である和として表現することで $O(\log k)$ で計算可能となりました。

18.2 部分木とパス - Subtrees and paths

木の探索順 - tree traversal array は根をもつ木のノードを、ルートノードからの深さ優先探索 で訪れる順番に並べたものです。



このような場合は深さ優先探索を行い、



このように辿るため、木の探索順は次のようになります。

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

部分木クエリ - Subtree queries

木における部分木とは木の探索順のある部分配列に対応して、その部分配列の最初の要素がその部分木の根となるようにします。

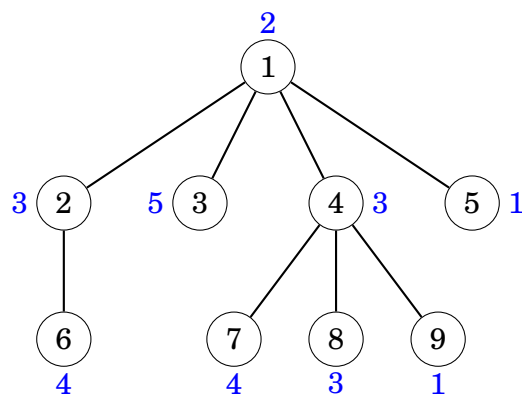
例えば、以下の部分配列は、ノード 4 の部分木となります。

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

これを利用して、木の部分木対するクエリを効率的に処理できます。各ノードに値があるとした時に、次のようなクエリを考えます。

- 単一ノードの値を更新する
- 単一の指定したノードの部分木の合計を計算する

次の図で青い数字がノードの値とします。例えば、ノード 4 の部分木の和は、 $3 + 4 + 3 + 1 = 11$ となります。



この問題を解くアイデアは各ノードに対して、ノードの番号、サブツリーのサイズ、ノードの値を持っておきます。次のようになります。

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

この配列を使って部分木の大きさを求め、次に対応するノードの値を求めれば任意の部分木の値の合計を計算することができます。ノード 4 の部分木の値は以下のように求めます。

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

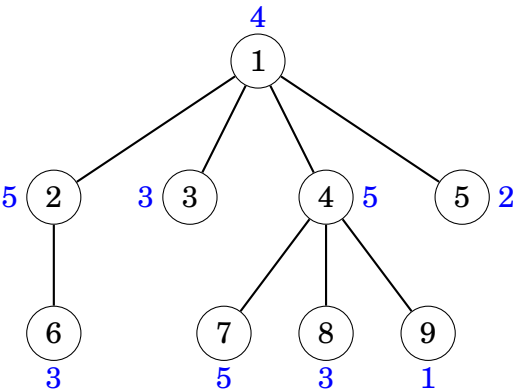
クエリに効率的に答えるにはこれを格納する構造体を工夫することが必要です。バイナリインデックストリー (BIT) またはセグメントツリーなどが適切でしょう。これらを用いると値の更新と値の総和の計算の両方を $O(\log n)$ で行うことができます。

パスクエリ - Path queries

また、木の探索順を用いると、根から木の任意のノードまでのパスの値の総和を効率的に計算することができます。次のようなクエリを考えましょう。

- 単一ノードの値を変更する
- 根から単一ノードへの値の総和を考える

例えば、次の図では根からノード 7 へのコストは $4+5+5=14$ となります。



これは先ほど同様に解くことができますが、今度は配列の最後の行の各値がルートからノードへのパス上の値の合計となります。

ノード番号	1	2	6	3	4	7	8	9	5
部分木のサイズ	9	2	1	1	4	1	1	1	1
値の合計	4	9	12	7	9	14	12	10	6

あるノードの値が x 増加した時、そのサブツリーの値は全て x 増加します。例えば、ノード 4 が 1 増加した時、以下のように変化が起こります。

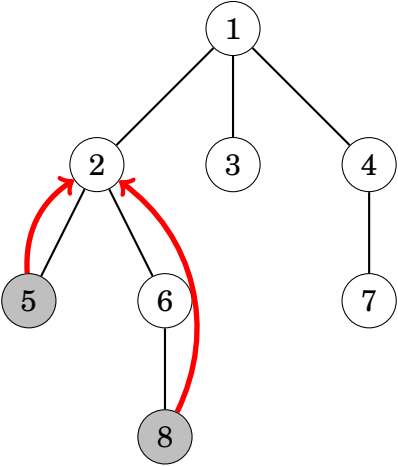
ノード番号	1	2	6	3	4	7	8	9	5
部分木のサイズ	9	2	1	1	4	1	1	1	1
値の合計	4	9	12	7	10	15	13	11	6

この 2 つの操作をサポートするためには、区間更新が可能である値が取り出すデータ構造が必要です。これは、バイナリインデックスまたはセグメントツリーを使用して $O(\log n)$ 時間で行うことができます (9.4 章参照)。

18.3 最小共通祖先 (LCA) - Lowest common ancestor

根付き木の 2 つのノードの**最小共通祖先 (LCA) - lowest common ancestor** とは、あるノードの部分木が 2 つのノードを含むような最も下のノードのことです (訳註: つまり対象のノードに最も近いノードです)。最も典型的な問題は 2 つのノードのペアが与えられるのでその問い合わせを効率的に行う問題でしょう。

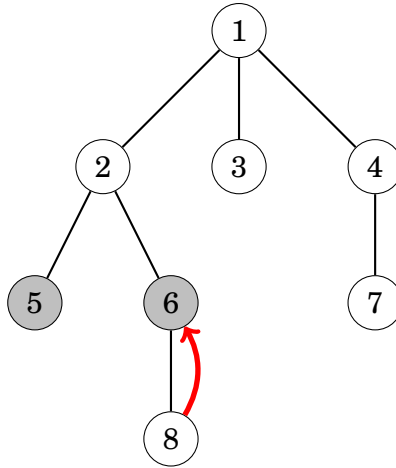
以下の木では、ノード 5 と 8 の LCA はノード 2 です。



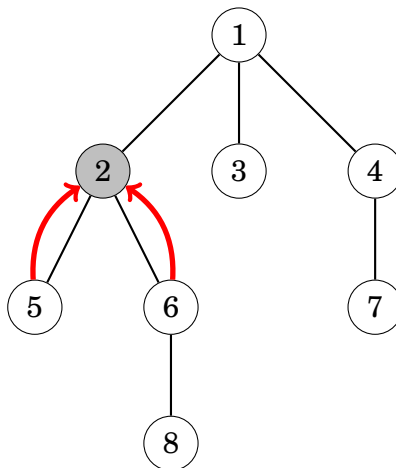
さて、LCA を求めるための 2 つの方法を紹介します。

手法 1: ダブリング - Method 1

1つめの方法では、先ほど紹介した木の任意のノードの k 番目の先祖は効率的に見つけることができる性質を利用します。これを利用して LCA を 2 つの問題として捉えられます。2 つのポインタを使い、最初は対象の 2 つのノードを指します。ここで深い方のポインタの一方を上方に移動させ、両方のポインタが同じ深さとなるようにします。以下の 2 番目のポインタを 1 レベル上げて、ノード 5 と同じ深さにあるノード 6 を指すようにします。



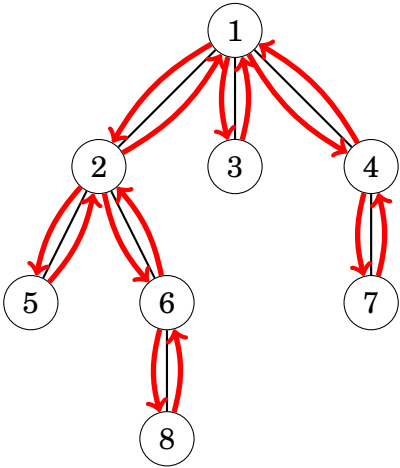
次に、両者のポインタを上方に移動させ、同じノードを指すようにするために必要な最小のステップ数を求めます。この後、ポインタが指すノードが LCA となります。この例では、両方のポインタを一段階上のノード 2(LCA) に移動させることになります。



このアルゴリズムは事前に計算された情報を使って $O(\log n)$ 時間で実行できるので、任意の 2 つのノードの LCA は $O(\log n)$ で見つけることができます。

手法 2: オイラーツアー - Method 2

この方法は木の探索順によって LCA を求めます。^{*1} これは深さ優先探索を有効に使う方法です。



ここでは先ほどと異なるのは別の木の探索配列を使っていることに注意します。深さ優先探索がノードを通過するときに最初の訪問した時間だけでなく、常に各ノードを配列に追加します。したがって、 k 個の子を持つノードは配列中に $k+1$ 回出現し、配列中には合計 $2n-1$ 個のノードが存在することになります。

配列には、ノードの識別子と、そのノードの木における深さを持ちます。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ノード番号	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
深さ	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

ここで、ノード a とノード b の LCA は、配列のノード a とノード b の間の深さが最小のノードです。例えば、ノード 5 と 8 の LCA は次のようになります。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ノード番号	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
深さ	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

ノード 5 は index 2 にあり、ノード 8 は index 5 にあり、index 2...5 の間の最小のノードを求めます。従って、ノード 5 とノード 8 の LCA は深さ 2 であるノード 2 となります。2 つのノードの LCA をを見つけるには、区間最小を求めるクエリを処

^{*1} This lowest common ancestor algorithm was presented in [7]. This technique is sometimes called the **Euler tour technique** [66].

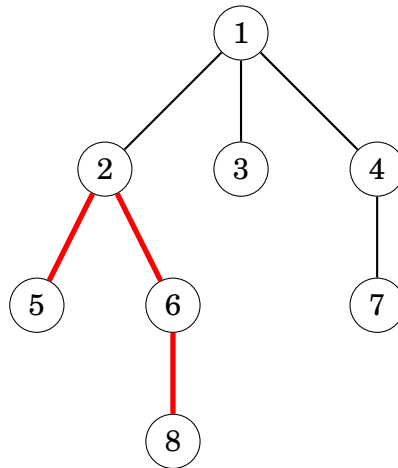
理すればよいです。配列は静的なので $O(n \log n)$ の前処理をした後、 $O(1)$ 時間でこのようなクエリを処理することができます。(訳注: 前述の通りに Sparse Table などを用いることでこれが実現できます)

ノード間の距離 - Distances of nodes

ノード a と b の距離とは a から b へのパスの長さです。ノード間の距離を計算する問題は、ノード間の LCA を見つけることが大切だとわかります。まず、木を根付き木として考えます。ノード a 、 b の距離は次の式で計算できます。

$$\text{depth}(a) + \text{depth}(b) - 2 \cdot \text{depth}(c),$$

ここで、 c は a と b の LCA ノード、 $\text{depth}(s)$ はノード s の深さです。例えば、ノード 5 と 8 の距離を考えてみます。



ノード 5, 8 の LCA は 2 です。ここで、 $\text{depth}(5) = 3$, $\text{depth}(8) = 4$ and $\text{depth}(2) = 2$, であるため、5, 8 の距離は $3 + 4 - 2 \cdot 2 = 3$ となります。

18.4 オフラインのアルゴリズム - Offline algorithms

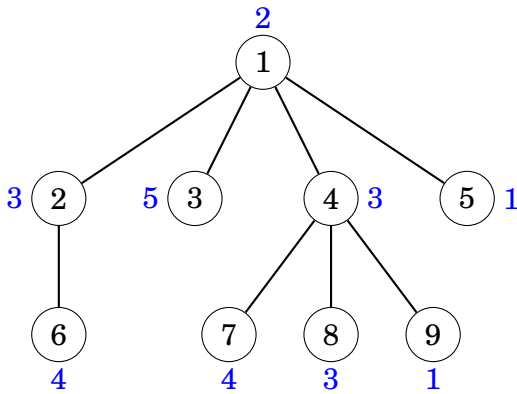
これまで私たちはオンライン - *online* な木クエリのアルゴリズムを考えてきました。

オンラインアルゴリズムは、次のクエリがくる前にそのクエリに応答するように、クエリを順番に処理することができます。これらが必要ないオフライン - *offline* なアルゴリズムに注目します。オフラインであるとは、任意の順序で回答可能な問い合わせのセットが与えられます。つまり、最初から全てのクエリが分かっているとします。一般にオンラインアルゴリズムと比較してオフラインアルゴリズムの設計は容易になります。

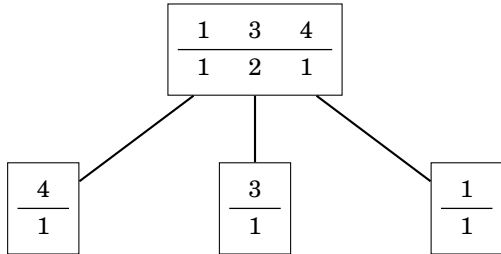
データ構造のマージ - Merging data structures

オフラインのアルゴリズムの一例として、深さ優先の木探索によりノードにデータ構造を保持する方法があります。各ノード s において s の子のデータ構造に基づくデータ構造 $d[s]$ を作成し、このデータ構造を用いて、 s に関連する全ての問合せを処理していくのです。

各ノードが何らかの値を持つ木があるとします。与えられた問題は「ノード s の部分木にある値 x を持つノードの数を計算せよ」だとします。例えば、以下の木において、ノード 4 の部分木には値が 3 であるノードが 2 つ含まれている。



この問題では、`map` 構造を用いてクエリに答えることができます。例えば、ノード 4 とその部分木のノードの `map` は以下の通りになります。



このようなデータ構造を各ノードに対して作成すれば、作成した直後にノードに関連するすべての問い合わせを処理することができ、与えられたすべての問い合わせを容易に処理することができます。例えば、ノード 4 に対する上記のマップ構造は、その部分木に値が 3 である 2 つのノードが含まれることがわかります。

ですが、これらすべてのデータ構造を一から作成するのは時間がかかりすぎます。そこで、各ノード s において、 s の値のみを含む初期データ構造 $d[s]$ を作成し、その後に s の子を訪ねて u が s の子であるすべてのデータ構造 $d[u]$ と $d[s]$ をマージすることにします。例えば、ノード 4 のマップは、以下のマップをマージして作成されています。



ここでは、最初のマップがノード 4 の初期データ構造と、他の 3 つのマップ がそれぞれノード 7、8、9 に対応しています。

ノード s でのマージは次のようにします。まず、 s の子を訪問し、各 u で $d[s]$ と $d[u]$ をマージする。つぎに、 $d[u]$ から $d[s]$ へは常に内容をコピーする。この前に、 $d[s]$ が $d[u]$ よりも小さければ、 $d[s]$ と $d[u]$ の内容を入れ替える。このようにすると、木の探索中に各値は $O(\log n)$ 回だけコピーされることになり、この アルゴリズムは効率的に動作します。2 つのデータ構造 a, b の内容を効率よく入れ替えるのは非常に簡単です。

```
swap(a,b);
```

a と b が C++ 標準ライブラリのデータ構造であるとき、上記のコードは定数時間で動作することが保証されています。

LCA - Lowest common ancestors

また、LCA のクエリを処理するオフラインのアルゴリズム^{*2}もあります。このアルゴリズムは union-find データ構造 (第 15.2 章参照) に基づいており、先ほどに説明したアルゴリズムよりも簡単に実装できます。

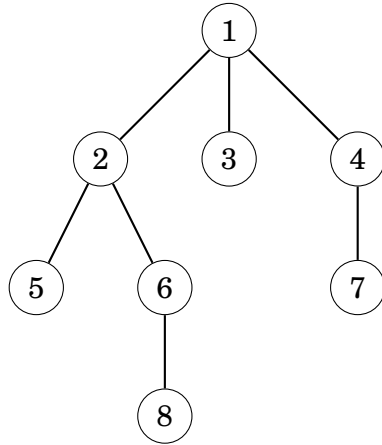
このアルゴリズムは、入力としてノードのペアのセットが与えられ両ノードの LCA を決定します。まず、各ノードが別の集合に属する木を作り、深さ優先の木探索を行います。初期状態では、各ノードは別々の集合に属している。また、各集合に対して、その集合に属する木における最も高さが高い（根に近い）ノードを保存しておきます。

このアルゴリズムでは、 x を訪問した時に、 y が既に訪問されているなら、 y の属する最も高いノードが LCA であると判断します。そして、ノード x を処理した後、アルゴリズムは x とその親の集合を結合します。

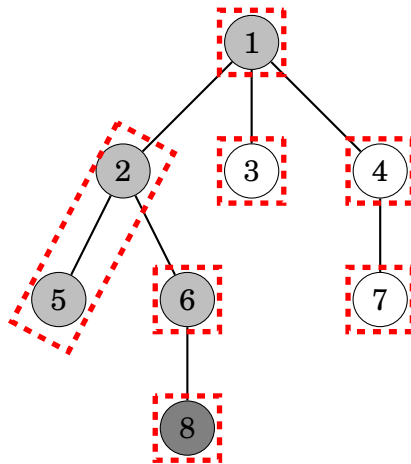
(訳注: このアルゴリズムはオイラーツアーを行い、戻りの処理の際に親の集合との結合を行います。そして、子側の集合の親は結合先の最も浅いノードだと情報を更新します。これは Union-Find の Leader に関して情報を持っておけば実装できます)

今、2 つのクエリ (5, 8), (2, 7) を受けこの LCA を求めたいとします。

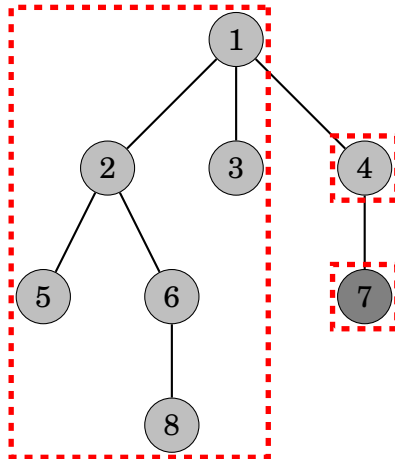
^{*2} This algorithm was published by R. E. Tarjan in 1979 [65].



次の図で灰色のノードは既に訪問したノードを表しており、破線で囲ったノードは同じ集合に属していることを示します。ノード 8 を訪問した時にノード 5 が既に訪問済みであり、その集合の中で最も高いノードは 2 であることに気づきます。したがって、ノード 5 と 8 の最小公倍数の祖先は 2 となります。



探索を進めノード 7 に到着した時が次の木です。同様に LCA は 1 であることがわかりました。



第 19 章

経路と閉路 - Paths and circuits

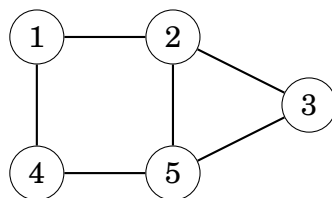
この章では次の 2 つの経路を扱います。

- オイラー路 - **Eulerian path** 各辺をちょうど 1 回ずつ通る経路
- ハミルトン路 - **Hamiltonian path** 各ノードをちょうど 1 回ずつ訪れる経路

一見、オイラー経路とハミルトン経路は似た経路に見えますが全く異なったものです。グラフにオイラー経路があるかどうかは非常に簡単なルールがあり、オイラー経路がある場合にそれを見つける効率的なアルゴリズムが存在します。しかし、ハミルトンパスの存在を確認することは、NP 困難 (NP-Hard) な問題で、この問題を解くための効率的なアルゴリズムは知られていません。

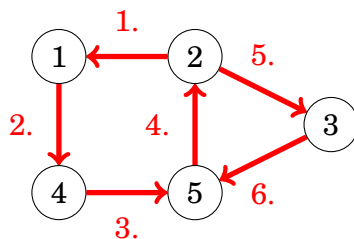
19.1 オイラー路 - Eulerian paths

オイラー路 - **Eulerian path**^{*1} はグラフの各辺をちょうど 1 回ずつ通る経路です。

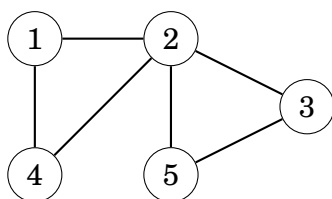


このグラフはノード 2 からノード 5 までのオイラー路を持ちます。

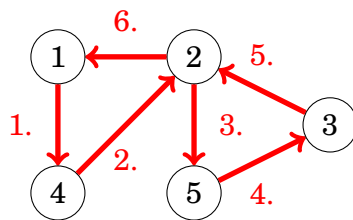
^{*1} L. Euler studied such paths in 1736 when he solved the famous Königsberg bridge problem. This was the birth of graph theory.



オイラー閉路 - Eulerian circuit とは、同じノードで始まり、同じノードで終わるオイラー路です。



このグラフでは以下のようなノード 1 開始のオイラー閉路を持ちます。



存在の判定 - Existence

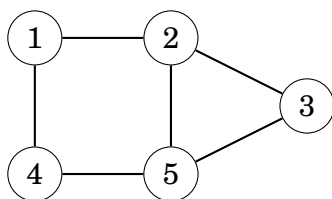
オイラー路と閉路の存在はノードの次数によって判定できます。無向グラフにおいてオイラー路を持つのはすべての辺が同じ連結成分に属し、

- 全てのノードの次数が偶数
- ちょうど 2 つのノードだけの次数が奇数で他のすべてのノードの次数が偶数

この 2 つのどちらか一方を満たしている場合である。

最初の場合に各オイラー路はオイラー閉路でもあります。

2 番目の場合は奇数次ノードの 2 つのオイラーパスの始点と終点になります。次のグラフを例にします。



ノード 1、3、4 は次数 2 でノード 2、5 が次数 3 で、ちょうど 2 つのノードが奇

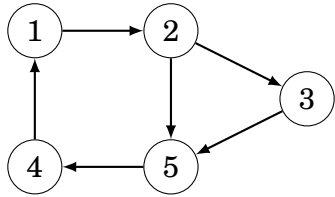
数の次数を持ちます。ノード 2 と 5 の間にはオイラー経路が存在するが、このグラフにはオイラー閉路は存在しません。

有向グラフでは、ノードの出次数と入次数に注目します。有向グラフにおいてオイラー路を正確に含むのは、すべての辺が同じ連結成分に所属しておりかつ、

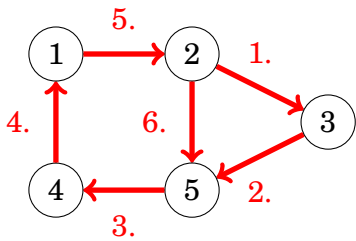
- 全てのノードにおいて出次数と入次数が等しい
- ある一つのノードで入次数が出次数より 1 だけ大きく、他の一つのノードで出次数が入次数より 1 だけ大きく、他の全てのノードでは出次数と入次数が等しい

この 2 つの条件のどちらかを満たしているグラフである必要があります。最初の条件を満たす場合、存在するオイラー路はオイラー閉路でもあります。2 つ目の条件を満たす場合、開始を出次数が大きなノードとして、終了を入次が大きなノードとしたオイラー閉路が存在します。

以下のようなグラフを考えます。



ノード 1、3、4 はともに入次数と出次数は 1 です。ノード 2 は入次数 1 で出次数は 2 で、ノード 5 は入次数 2 で出次数は 1 です。このため、次のようにノード 2 からノード 5 へのオイラー路が存在することがわかります。



ヒールホルツァーのアルゴリズム - Hierholzer's algorithm

ヒールホルツァーのアルゴリズム - Hierholzer's algorithm^{*2} は、オイラー閉路を構築する効率的なアルゴリズムです。このアルゴリズムはグラフに閉路が存在することを前提として、複数のラウンドを実行し、各ラウンドは閉路となる新しい辺を追加していきます。

^{*2} The algorithm was published in 1873 after Hierholzer's death [35].

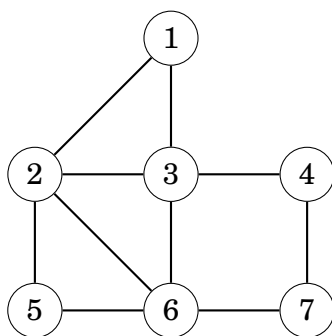
まず、グラフの辺の一部 (全てでなくてもよい) を含む回路を構築します。その後、この閉路に別の閉路を追加することにより、段階的に回路を拡張していきます。これをすべての辺が閉路に追加されるまで繰り返します。

このアルゴリズムは閉路に属しているノードで閉路に含まれていない出次辺を持つノード x を処理して閉路を拡張していきます。このノード x から、まだ閉路に含まれていないエッジのみを含む新しい経路を構築します。いずれそのパスはノード x に戻り、サブ閉路が作られる。

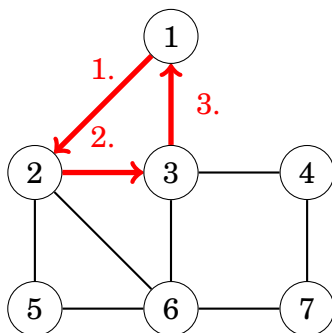
もし、グラフにオイラー路しかない場合でも、グラフに余分な辺を追加し、閉路を構築した後にその辺を削除すれば、ヒールホルツァーのアルゴリズムで路を求めることができます。例えば、無向グラフの場合は 2 つの奇数次ノードの間に余分なエッジを追加します

実際にヒールホルツァーのアルゴリズムが無向グラフに対し、どのようにオイラー路を構築するかを見ていきましょう。

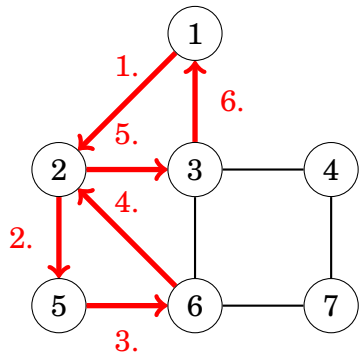
Example



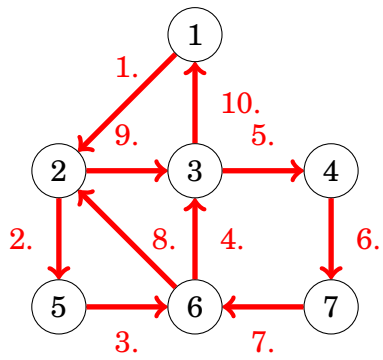
まずノード 1 から始まる回路を作成するとします。たとえば $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ です。



このあと、部分閉路である $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ を追加していきます。



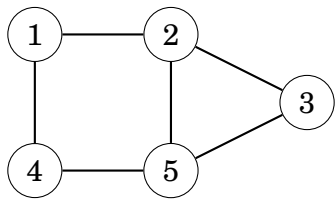
そして、 $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ を追加します。



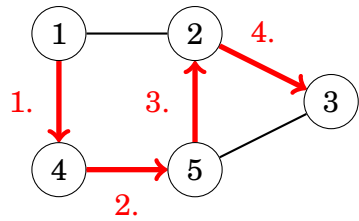
これによってオイラー閉路が完成しました。

19.2 ハミルトン路 - Hamiltonian paths

ハミルトン路 - Hamiltonian path というのは各ノードをちょうど1回ずつ訪問するパスのことです。

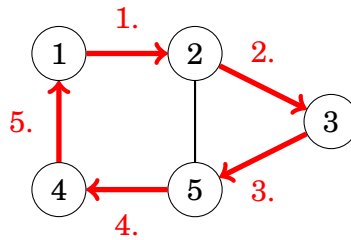


これはノード1からノード3 というハミルトン路があります。



ハミルトン路が同じノードで始まり、同じノードで終わるような経路をハミルト

ン閉路と呼びます。上のグラフにはノード 1 から始まりノード 1 で終わるハミルトン閉路があります。



存在の確認

グラフにハミルトン路が含まれるかどうかを調べる効率的な方法は知られておらず、この問題は **NP-Hard** です。しかし、いくつかのケースでグラフがハミルトン路を含んでいることがわかっています。

単純な考察からわかるのはグラフが完全である場合、すなわちすべてのノードのペアの間にエッジがある場合にはハミルトン路も含んでいることになります。さらにいくつかの証明もされています。

- **ディラックの定理 - Dirac's theorem:** 全てのノードが $n/2$ 以上の次数である場合、グラフにはハミルトン路が存在する
- **オレの定理 - Ore's theorem:** 隣接しない全てのペアの次数の和が n 以上である場合、グラフにはハミルトン路が存在する

これらに共通する性質は、グラフが多数の辺を持つ場合にハミルトンパスの存在を保証しています。これは、グラフに含まれる辺の数が多いほど、ハミルトン路を構成する可能性が高いので理にかなっています。

構築 - Construction

そもそもハミルトン路が存在するかどうかを判定する効率的な方法がないため、パスを効率的に構築する方法もありません。パスが構築できるなら、それが存在するかどうかを確認すればよいです。

さて、ハミルトンパスを探索する最もシンプルな方法は、パスを構成する可能なすべての方法を調べるバックトラックのアルゴリズムを使用することです。ただし、このアルゴリズムの時間計算量は少なくとも $O(n!)$ です。

より効率的な解決策は動的計画法に基づいていて実現できます (10.5 章参照)。このアイデアは、関数 $\text{possible}(S, x)$ の値を計算することです。ここで S はノードの部分集合、 x はノードの 1 つとします。この関数は、 S のノードを訪れ、ノード x で

終わるハミルトンパスが存在するかどうかを判定します。この解法は $O(2^n n^2)$ 時間で実装できます。

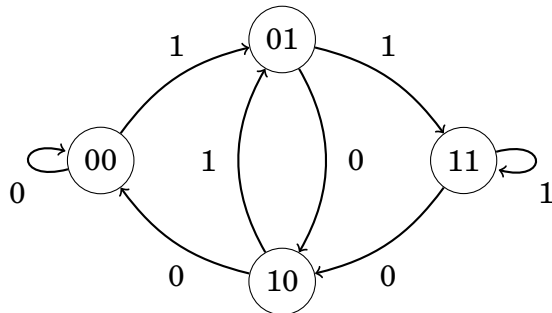
19.3 デ・ブルーイエン配列 - De Bruijn sequences

デ・ブルーイエン配列 - De Bruijn sequence は、文字数 k の固定アルファベットの文字列に対して、長さ n のすべての文字列を部分文字列としてちょうど一度だけ含む文字列のことです。このような文字列の長さは $k^n + n - 1$ 文字です。例えば、 $n = 3, k = 2$ のときに De Bruijn 配列の例は次のようになります。

0001011100.

これは以下のような全ての部分文字列を含みます。000, 001, 010, 011, 100, 101, 110, 111.

De Bruijn 配列は、グラフのオイラー路に対応することが分かっています。そこで、各ノードが $n - 1$ 文字の文字列を含み、各辺がその文字列に 1 文字ずつ追加するグラフを作成します。次のグラフは、上記のシナリオに対応するグラフです。



このグラフのオイラー路は、長さ n の文字列をすべて含む文字列に対応します。この文字列には開始ノードの文字とエッジの文字がすべて含まれています。始点ノードは $n - 1$ 文字でエッジには k^n 文字があるため、文字列の長さは $k^n + n - 1$ となります。

19.4 ナイトツアー - Knight's tours

ナイトツアー - knight's tour は $n \times n$ のチェス盤上で、チェスのルールに従って騎士(ナイト)が各マスにちょうど 1 回ずつ訪れるような動きのことです。ナイトツアーは、最終的にスタート地点のマスに戻る場合はクローズドツアー、そうでない場合はオープンツアーと呼ばれます。例えば、 5×5 ボードのオープンナイトのツアーを示します。

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

ナイトツアーは、盤上のマスをもとにしたグラフのハミルトン路といえます。ナイトがチェスのルールに従ってマスの間を移動できる場合、2つのノードを辺で結びます。

ナイトツアーを構成する一般的な方法はバックトラックを使うことです。完全なツアーを素早く見つけるためにはヒューリスティックを用いると効率的です。

ウォーンズドルフの法則 - Warnsdorf's rule

ウォーンズドルフの法則^{*3}は、ナイトのツアーを見つけるためのシンプルで効果的な手法です。この法則を用いると、 n が大きな盤面でも効率的にツアーを構成することができます。この法則は移動可能なマスの数ができるだけ少なくなるように、常にナイトを移動させていきます。

次のような状況で、ナイトが移動できるマスは5つあります (squares $a \dots e$)。

1				a
		2		
b				e
	c		d	

この場合、ウォーンズドルフのルールでは、ナイトを a マスに移動させます。この選択の後は1手しかないためです。他の選択肢では3手まで可能なマスにナイトを移動させることができます。

^{*3} This heuristic was proposed in Warnsdorf's book [69] in 1823. There are also polynomial algorithms for finding knight's tours [52], but they are more complicated.

第 20 章

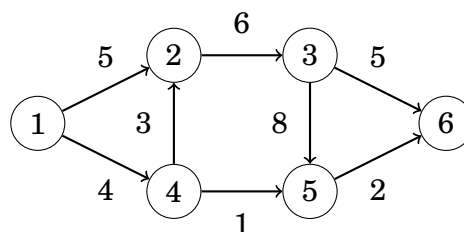
フローとカット - Flows and cuts

この章では以下の 2 つの問題について解説します。

- **最大流の発見 - Finding a maximum flow:** あるノードから他のノードに送ることができるフローの最大の量は？
- **最小カットの発見 - Finding a minimum cut:** グラフを二分割する辺のカットを最小にするには？

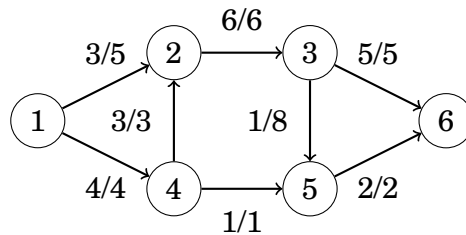
これらを考えるグラフは 2 つの特殊なノードを含む重み付き有向グラフです。ソースは入次辺を持たないノードでシンクは出次辺を持たないノードとします。

次のグラフではノード 1 がソース、ノード 6 がシンクです。



最大流 - Maximum flow

最大流 - maximum flow を求めるタスクはソースからシンクにできるだけ多くのフローを送ることです。各エッジの重みはそのエッジを通過できる最大のフローで容量と呼ばれます。各中間ノードにおいて流入するフローと流出するフローは等しくなければなりません。次のグラフの例ではフローの最大サイズは 7 で、そのフローをどのようにルーティングするかを示しています。

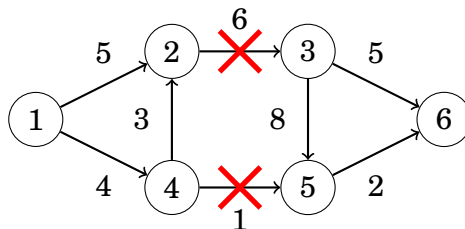


v/k という表記は、 v の流量のフローが k 単位の容量を持つエッジを経由していることを意味します。送信元が $3+4$ のフローを送信し、シンクが $5+2$ ユニットのフローを受信するので、グラフのフローの大きさは 7 です。このグラフではシンクにつながるエッジの総容量は 7 なので、このフローが最大であることは容易に理解できるでしょう。

最小カット - Minimum cut

最小カット - minimum cut 問題とは、グラフから辺の集合を削除していき、削除後にソースからシンクへのパスが存在しなくなり、削除した辺の総コストが最小となるようなものです。

例題のグラフのカットの最小サイズは 7 であり、辺 $2 \rightarrow 3$ 、 $4 \rightarrow 5$ を削除すれば良いです。



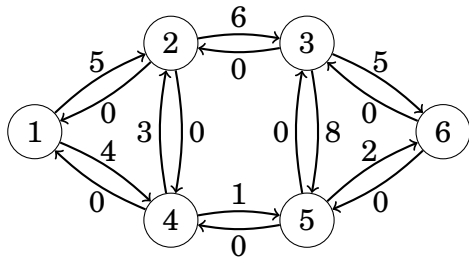
エッジを削除すると、ソースからシンクへのパスは存在しなくなります。削除されたエッジの重みは 6 と 1 であるため、このカットの大きさは 7 です。

上の例で最大流とカットの最小サイズが同じになりましたがこれは偶然ではありません。最大フローと最小カットは常に同じ大きさです。それでは、グラフの最大フローと最小カットを求めるためのフォード・ファルカーソンのアルゴリズムを見ていきます。このアルゴリズムを理解することで最大流と最小カットがなぜが等しくなるのか理解できるでしょう。

20.1 フォード・ファルカーソンのアルゴリズム - Ford - Fulkerson algorithm

フォード・ファルカーソンのアルゴリズム - Ford - Fulkerson algorithm [25] は、グラフの最大流量を求めるアルゴリズムです。空のフローから始め、各ステップではより多くのフローを生成するソースからシンクへの経路を見つけます。これ以上フローを増やせなくなったときに最大フローが発見されたと判定します。このアルゴリズムは、元の辺がそれぞれ別の方向の逆辺を持つという特殊な表現を使います。各エッジの重みはそのエッジを経由してどれだけ多くのフローをルーティングできるかを示します。アルゴリズムの開始時点では各オリジナルエッジの重みはエッジの容量に等しく、各逆エッジの重みはゼロとしておきます。

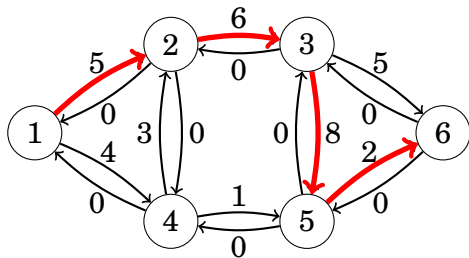
これを示したグラフは次のようになります。



アルゴリズムの説明 - Algorithm description

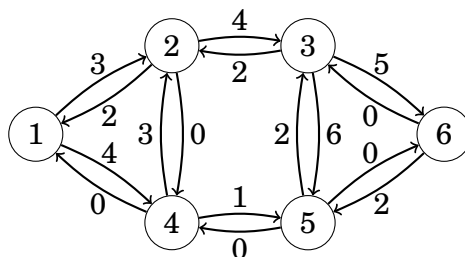
この処理は複数のラウンドで構成されます。各ラウンドで、アルゴリズムはソースからシンクへのパスを見つけ、そのパス上の各エッジが正の重みを持つようにします。複数の可能な経路がある場合は、そのうちのどれかを選択します。

例えば、次のようなパスを選択したとしましょう。



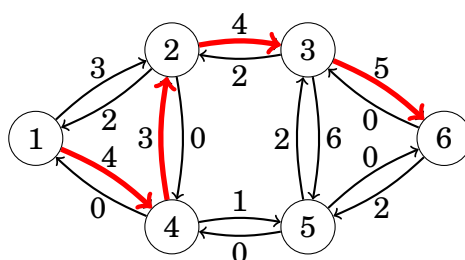
経路を選択した後、フローはその経路に流せる最大の量である x 単位で経路を増やします。ここで、 x は経路上の最小のエッジの重みとなります。また、経路上の各辺の重みを x だけ減少させ、各逆辺の重みを x だけ増加させます。上のパスでは、エッジの重みは 5、6、8、2 です。最も小さい重みは 2 なので、フローは 2 だ

け増加し、新しいグラフは以下ようになります。



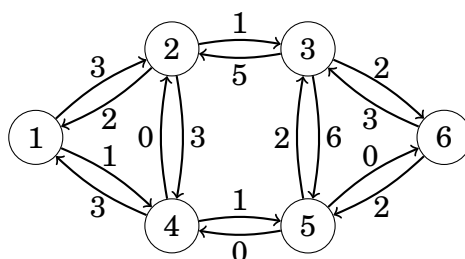
つまり、フローを増やすとそのエッジを通ることができるフローが減少するという操作です。一方、フローを別の方法で流した方が有益であることが判明した場合、グラフの逆エッジを使用してそのフローを後でキャンセル（減らす）ことができます。

このアルゴリズムは、正の重みの辺を通るソースからシンクへの路が存在する限り、操作を繰り返してフローを増加させます。先ほどの直後では、次のような路が考えられる。

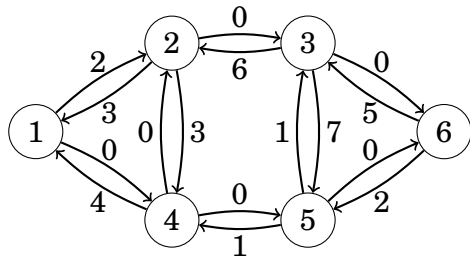


このパスの最小の重さは3なので、このパスによってフローは3増加し、パス処理後の総フローは5となります。

新しいグラフは以下ようになります。



最大流量に達するまで、まだ2ラウンド操作できます。例えば、 $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ と $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$ という経路です。どちらの経路もフローを1増加させ、最終的なグラフは以下ようになります。



ソースからシンクまで正のエッジ重みを持つパスが存在しないため新しい路はつくれません。アルゴリズムは終了し、最大フローは7となりました。

パスの検索 - Finding paths

フォード・ファルカーソン アルゴリズムでは、流量を増加させる経路をどのように選択すべきかは指定されていません。どのような路を選んでも、アルゴリズムは遅かれ早かれ収束し、正しく最大流量を求めることができます。ただし、経路の選び方次第で効率は変わってきます。

経路を見つける簡単な方法は深さ優先探索です。通常の場合でこれはうまくいきますが、最悪のケースでは各パスがフローを1だけしか増加させないのでアルゴリズムは遅く動作します。そこで次のような手法を用いることで効率的にパスを選択できます。

エドモンズ・カーブのアルゴリズム - Edmonds - Karp algorithm [18] は、なるべく辺の数ができるだけ少なくなるようにパスを選択します。これは、パスには幅優先探索を用いることで実現できます。これにより、フローが迅速に増加することが保証され、このアルゴリズムの時間複雑性は $O(m^2n)$ であることが証明され得る。

スケーリングアルゴリズム [2] は深さ優先探索により、選択する各辺の重みが少なくとも閾値以上である経路を見つけるものです。初期状態では閾値はある大きな数、例えばグラフの全てのエッジ重みの和からスタートします。パスが見つからないときは閾値を2で割ります。このアルゴリズムの時間計算量は $O(m^2 \log c)$ であり、ここで c は初期の閾値です。深さ優先探索で経路を見つけることができるためスケーリングアルゴリズムの方が実装が簡単です。

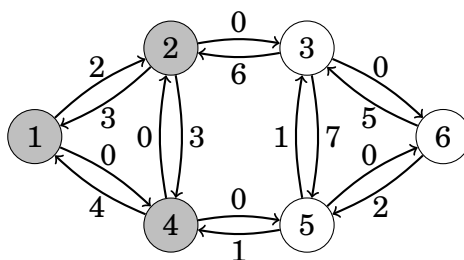
どちらのアルゴリズムもプログラミングコンテストでよく出題されるような問題には十分です。

最小カット - Minimum cuts

フォード・ファルカーソンアルゴリズムが最大流量を求めると、最小カットも求められることがわかっています。正の重みのエッジを使ってソースから到達可能な

ノードのある集合を A とします。

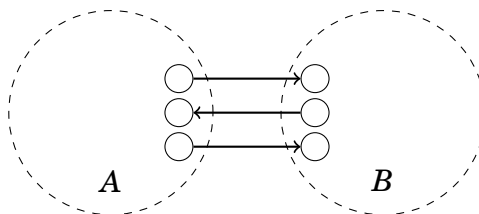
例のグラフでは A にはノード 1、2、4 が含まれています。



最小カットは元のグラフの A 内のあるノードから始まり A 以外のノードで終わり、その容量は最大フローで完全に使われる辺で構成されています。上のグラフでは、 $2 \rightarrow 3$ と $4 \rightarrow 5$ がそのような辺で、最小カット $6+1=7$ に相当します。

アルゴリズムが生成するフローはなぜ最大流となり、カットはなぜ最小となるのでしょうか？ その理由は、グラフのカットの重さよりも大きなサイズのフローというのは含むことができないからです。したがって、フローとカットが同じ大きさであれば常に最大フローと最小カットとなります。

例としてソースが A に属し、シンクが B に属し、その集合の間にいくつかのエッジが存在するようなグラフのカットを考えることにしましょう。



カットのサイズとは A から B に進む辺の合計です。これはグラフ内のフローが A から B に流れているということです。したがって、最大流の大きさは、グラフのどのカットの大きさよりも小さいか等しくなることは明らかです。

一方、フォード・ファルカーソン法ではグラフのカットのサイズとちょうど同じ大きさのフローが生成されます。したがって、フローは最大フローでなければならない、カットは最小カットでなければならないのです。

20.2 素なパス - Disjoint paths

多くのグラフ問題は最大流問題に還元することで解くことができます。1つの例をあげます。

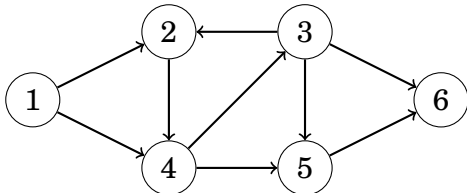
ソースとシンクを持つ有向グラフが与えられた時に、ソースからシンクへの素な

パスの最大数を見つけてください。

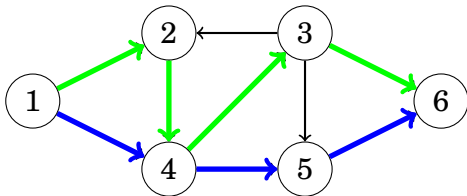
辺素なパス

ソースからシンクまでの辺素なパスの最大数を求めましょう。つまり、各エッジが最大でも 1 つの経路に現れるような経路の集合を構成します。

例えば、次のようなグラフを考えてみましょう。



このグラフにおいて、辺素なパスの最大数は 2 で、 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ と $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ を以下のように選ぶことができます。

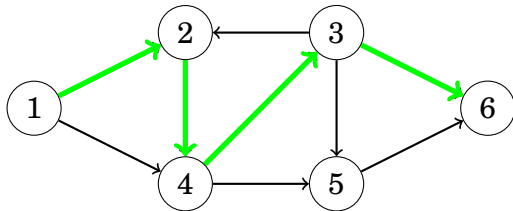


各エッジの容量を 1 とすると、辺素なパスの最大数はグラフの最大フローに等しいことが判明しました。最大流が構成された後の辺素なパスの経路は、ソースからシンクへの経路を貪欲にたどることで見つけることができます。

ノードが素なパス - Node-disjoint paths

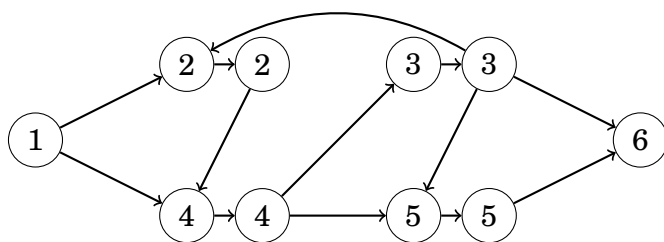
では別の問題として、ソースからシンクまでのノードが素なパスの最大数を求めます。ただし、ソースとシンクは複数回登場しても良いです。

このグラフの場合は 1 となります。

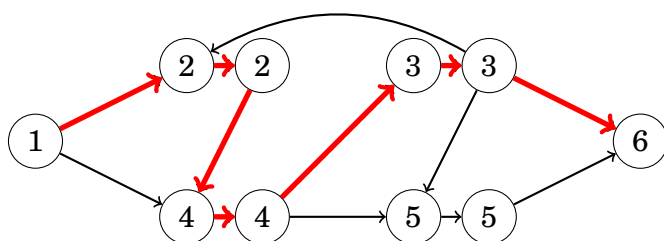


この問題も最大流問題とすることができます。各ノードは最大でも 1 つの経路にしか現れないので、ノードを通過する流れを制限しなければなりません。このための標準的な方法は、各ノードを 2 つのノードに分割し、最初のノードは元のノードの入次辺を持ち、2 番目のノードが出次辺を持つようにします。そして、最初の

ノードから 2 番目のノードに向かう新しいエッジが存在するようにします。
この例では、グラフは次のようになります。



最大流は以下のようになります。



この最大流は 1 となり、このため、ノードが素な経路の数は 1 となります。

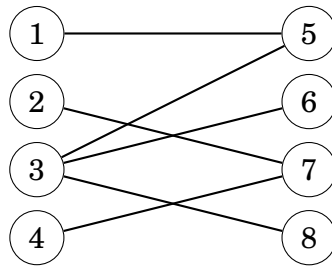
20.3 最大マッチング - Maximum matchings

最大マッチング問題 - maximum matching は、無向グラフにおいて各ノードの集合のペアを考え、各ノードが最大 1 つのペアに属するような最大サイズのノードペアの集合を求める問題です。

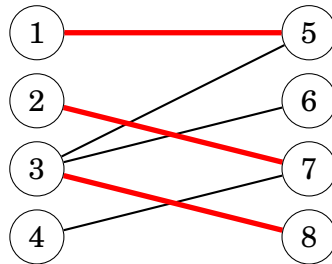
一般的なグラフでも最大マッチングを求める多項式アルゴリズムがありますが [17]、そのようなアルゴリズムは複雑でプログラミングコンテストで見られることはほぼありません。しかし、二部グラフでは最大マッチング問題は最大フロー問題に帰着できます。

最大マッチングの検出 - Finding maximum matchings

二部グラフのノードは常に 2 つのグループに分けられ、グラフのすべてのエッジが左のグループから右のグループに辺が張られています。例えば、次の二部グラフでは、グループは $\{1, 2, 3, 4\}$ と $\{5, 6, 7, 8\}$ です。

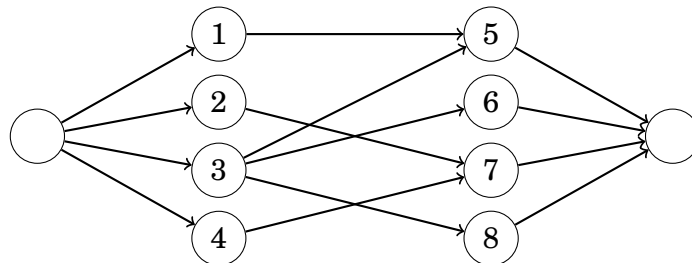


このグラフの最大マッチングの大きさは3です。

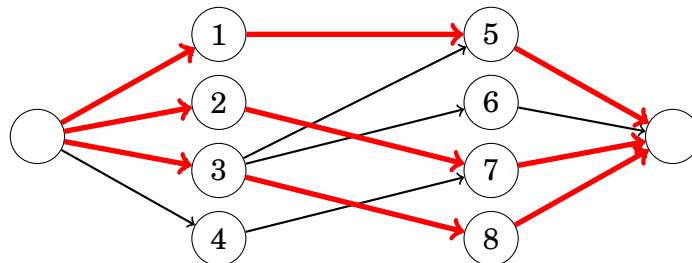


このグラフにソースとシンクという2つのノードを新たに加えることで、二部最大マッチング問題を最大フロー問題に還元することができます。ソースから各左ノードへ、各右ノードからシンクへのエッジを追加しましょう。この後、グラフ内の最大流の大きさは、元のグラフの最大マッチングの大きさと等しくなります。

次のようなグラフとします。



最大流は以下のようになります。



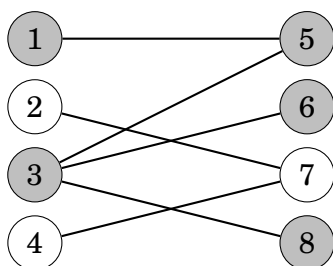
ホールの定理 - Hall's theorem

ホールの定理 - **Hall's theorem** は、二部グラフがすべての左ノードまたは右ノードを含むマッチングを持つかどうかを調べることができます。左右のノードの

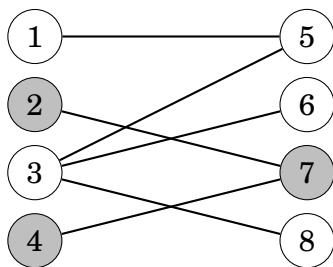
数が同じとき、グラフのすべてのノードを含む**完全マッチング**を構成できるかどうか、ホールの定理で判定できます。

すべての左ノードを含むマッチングを見つけたいとします。 X を任意の左ノードの集合、 $f(X)$ をその隣接ノードの集合としましょう。ホールの定理によれば、すべての左ノードを含むマッチングは、各 X について $|X| \leq |f(X)|$ の条件が成立するときに存在します。

ホールの定理を例のグラフで確認します。まず、 $X = \{1, 3\}$ とすると、 $f(X) = \{5, 6, 8\}$ です。



$|X| = 2$ かつ $|f(X)| = 3$ なので、条件は成り立っています。次に、 $X = \{2, 4\}$ で $f(X) = \{7\}$ です。



この場合、 $|X| = 2$ で $|f(X)| = 1$ なのでホールの定理の条件は成立しません。つまり、この例ではグラフの最大マッチングは 4 ではなく 3 であることがすでに分かっているとおりグラフの完全マッチングを形成することはできません。

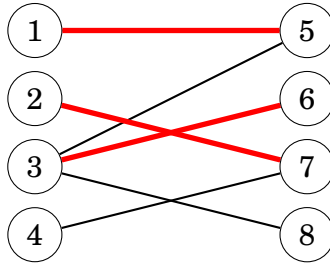
なぜこのようになるのか？ マッチングが成立しない理由を集合 X で説明しましょう。 X は $f(X)$ よりも多くのノードを含むので X のすべてのノードに対応するペアは存在しません。例えば、上のグラフではノード 2 とノード 4 はともにノード 7 と接続されて欲しいですが、それは不可能です。

ケーニヒの定理 - Knig's theorem

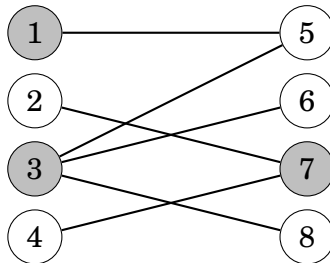
最小点被覆 - minimum node cover は、選択した点の各辺が全ての辺をカバーするようなノードの最小集合です。一般的なグラフでは、点被覆の最小値を求めることは NP-Hard な問題となっています。しかし、グラフが二部グラフであれば、

ケーニヒの定理により、最小点被覆の大きさは最大マッチングと等しくなります。このため、最大流のアルゴリズムでこれを求めることができます。

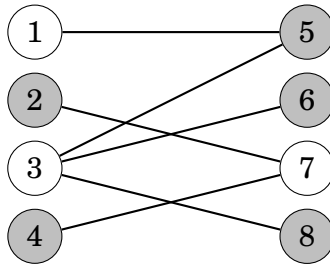
では、次のような最大マッチング 3 のグラフを考えます。



ケーニヒの定理から最小点被覆のサイズが 3 であることが分かります。



最小点被覆に属さないノードは、最大独立集合を形成します。これは、集合内の 2 つのノードがエッジで接続されていないようなノードの可能な限り大きな集合である。繰り返しますが、一般のグラフで最大独立集合を求めるのは NP-Hard な問題ですが、二部グラフではケーニヒの定理を使って効率的に問題を解くことができます。例のグラフでは、最大独立集合は次のようになります。



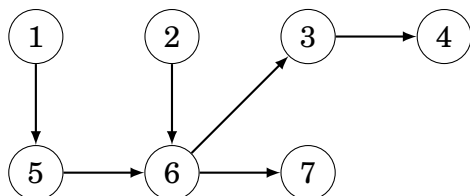
20.4 辺被覆 - Path covers

辺被覆 - path cover はグラフの各ノードが少なくとも 1 つのパスに属するようなグラフのパスの集合のことです。閉路を含まない有向グラフにおいて、最小辺被覆を求める問題は、別のグラフで最大流を求める問題に帰着できることがわかります。

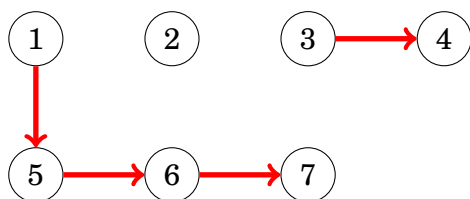
点素な辺被覆 - Node-disjoint path cover

点素な辺被覆 - **node-disjoint path cover**, では各ノードは正確に 1 つのパスに属します。

次のようなグラフを考えます。



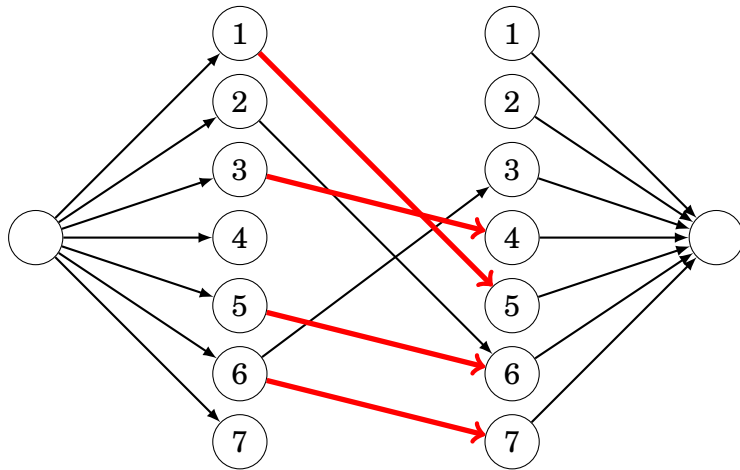
このグラフの最小の点素な辺被覆は、3 つのパスから構成されます。



なお、ノード 2 しか含まない集合があるようにパスに点が含まれないこともあります。

ここで、元のグラフの各ノードを左ノードと右ノードの 2 つのノードで表現したマッチンググラフを構築することで、最小の点素な辺被覆を求めることができます。元のグラフに左ノードから右ノードへのエッジがある場合、そのエッジはマッチンググラフにも存在します。また、マッチンググラフにはソースとシンクがあり、ソースからすべての左ノードへ、すべての右ノードからシンクへのエッジが存在します。

この結果得られたグラフの最大マッチングは元のグラフの最小の点素な辺被覆に対応します。上記のグラフに対する以下のマッチンググラフはサイズ 4 の最大マッチングを含んでいます。

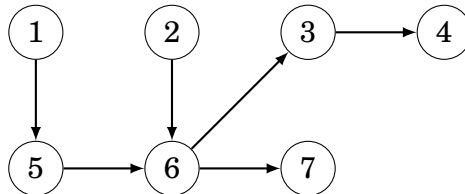


マッチンググラフの最大マッチングの各エッジは、元のグラフの最小の点素な辺被覆のエッジに対応します。したがって、最小の点素な辺被覆の最小サイズは $n - c$ といえます。ここで n は元グラフのノード数、 c は最大マッチングのサイズです。

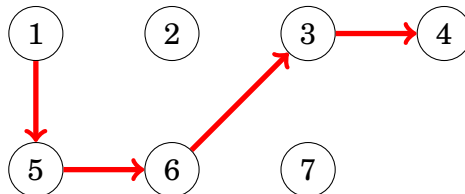
一般的な辺被覆 - General path cover

一般的な辺被覆 - **general path cover** は 1 つのノードが複数のパスに属することができる辺被覆のことです。1 つのノードは複数回パスで使われることがあるために、最小の一般的な辺被覆は最小の点素な辺被覆より小さくなることがあります。

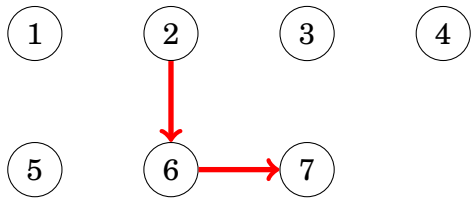
先ほどのグラフを例にします。



このグラフにおける最小の一般的な辺被覆は、2つの経路で作れます。まず1つ目は

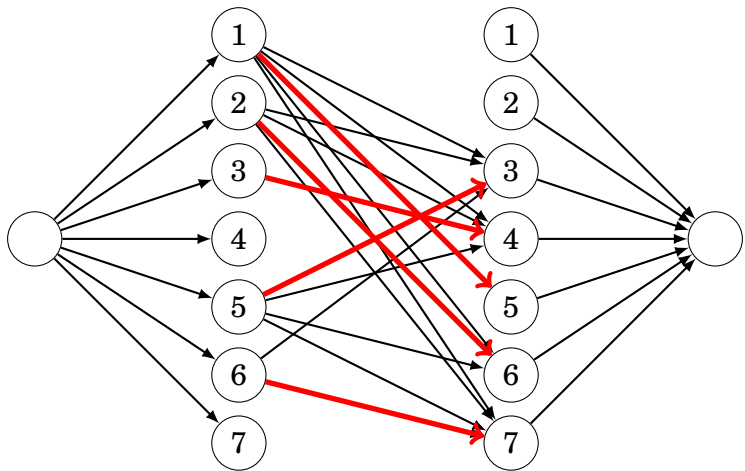


次に2つ目は、



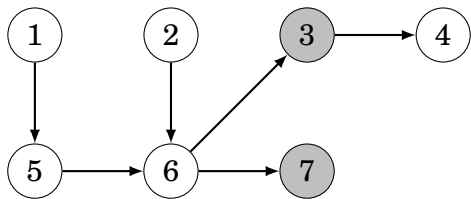
最小の一般的な辺被覆は、最小の点素な辺被覆とほぼ同様に求めることができます。マッチンググラフに新しい辺をいくつか追加し、元のグラフに a から b への経路があるときに必ず辺 $a \rightarrow b$ が存在するようにすればよいです (これはいくつかの辺を経由する場合があります)。

上記のグラフのマッチンググラフは次のようになります。



ディルワースの定理 - Dilworth's theorem

反鎖 - **antichain** とは、グラフの辺を使い、どのノードからも他のノードへの経路が存在しないようなノードの集合のことである。ディルワースの定理 - **Dilworth's theorem** では、閉路の存在しない有向グラフにおいて、最小の一般的な辺被覆の大きさは最大の反鎖の大きさに等しいとする。例えば、以下のグラフでは、ノード 3 と 7 が反鎖を形成しています。



これは最大反鎖となります。3つのノードを含むような反鎖は構成できないからです。このグラフの最小の一般的な辺被覆の大きさは、2つのパスからなることは先に確認しました。

第 III 部

発展的なテーマ - Advanced topics

第 21 章

整数論 - Number theory

整数論 (Number theory) は整数を研究する数学の分野です。整数論は一見単純そうに見えるかもしれませんが、整数に関わる多くの問題は非常に難しく、ですが興味深い分野です。

例として、次のような式を考えてみましょう。

$$x^3 + y^3 + z^3 = 33$$

実数の範囲では x, y, z の例を見つけるのは簡単で、例えば以下のものが思い浮かびます。

$$\begin{aligned} x &= 3, \\ y &= \sqrt[3]{3}, \\ z &= \sqrt[3]{3}. \end{aligned}$$

ところが、**整数 (integers)** である x, y and z を見つけるというのは未解決問題です。
[6]

この章では、整数論の基本的な概念とアルゴリズムに焦点を当てていきます。この章では特に断らない限り、すべての数は整数であると仮定します。

21.1 素数と因数 - Primes and factors

a が b の因数であるとき、 $a \mid b$ と表記し、そうでなければ $a \nmid b$ と記載します。

例えば 24 の因数は 1, 2, 3, 4, 6, 8, 12, 24 です。

$n > 1$ である数が 1 または n 以外の因数を持たない場合**素数**と呼びます。例えば、7、19、41 は素数ですが、35 は $5 \cdot 7 = 35$ なので素数ではありません。 $n > 1$ である数には、**素因数分解 - prime factorization** が唯一に存在します。

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

ここで、 p_1, p_2, \dots, p_k は異なる素数であり、 $\alpha_1, \alpha_2, \dots, \alpha_k$ は正の数です。84 の素因数分解は次のようになります。

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

ある数 n の因数の個数を

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

とします。なぜなら、各素数 p_i が因数に何回現れるかを選ぶ方法は $\alpha_i + 1$ 通りあるからです。例えば、84 の因子の数は、 $\tau(84) = 3 \cdot 2 \cdot 2 = 12$ です。実際に因数分解すると、1、2、3、4、6、7、12、14、21、28、42、84 となります。

n の **因数の和 - sum of factors** は

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

ここで、後者の式は幾何級数式 (TODO: geometric progression formula) によります。例えば、84 の因数の和は次の通りとなります。

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

n の **因数の積 - product of factors** は

$$\mu(n) = n^{\tau(n)/2},$$

となります。TODO: because we can form $\tau(n)/2$ pairs from the factors, each with product n . 84 の因数の積は次のようなペアで示されます。1・84, 2・42, 3・28 などなど。この結果、因数の積は $\mu(84) = 84^6 = 351298031616$ となります。

$n = \sigma(n) - n$ の場合、 n は**完全数 (perfect number)** と呼ばれます。これは n が 1 から $n-1$ までの因数の和に等しい場合です。例えば、28 は、 $28 = 1 + 2 + 4 + 7 + 14$ なので、完全数です。

素数は無限に存在する - Number of primes

素数が無限にあることを示します。もし、素数の数が有限であれば、素数全ての集合である $P = \{p_1, p_2, \dots, p_n\}$ を作ることができ、これはすべての素数を含みます。例えば、 $p_1 = 2, p_2 = 3, p_3 = 5$ です。ところが P 全ての要素を使って以下の P に含まれるのよりも大きな素数を作ることができます。

$$p_1 p_2 \cdots p_n + 1$$

これは矛盾するため、素数の数は無限となります。

素数の密度 - Density of primes

素数の密度とは、数字の中にどれくらいの頻度で素数が含まれているかのことで、 $\pi(n)$ を 1 から N までに含まれる素数の数とします。例えば、 $\pi(10) = 4$ です。1 から 10 までの間に 2、3、5、7 の 4 つの素数があることを意味します。

これは次のように近似的に示すことができます。

$$\pi(n) \approx \frac{n}{\ln n},$$

この式からわかる通り素数の密度は高いです。例えば 1 から 10^6 の実際の素数の数は $\pi(10^6) = 78498$ であり、 $10^6 / \ln 10^6 \approx 72382$ でおよそ一致します。

予想 - Conjectures

素数に関する **予想 (conjectures)** がいくつも存在します。これらのほとんどが正しいとされていますが証明はされていません。例えば以下のような予想が有名です。

- **ゴールドバッハ予想 - Goldbach's conjecture**: $n > 2$ は偶数である場合、 a および b がともに素数であるような $n = a + b$ の和で表すことができる。
- **双子の予想 (TODO) (Twin prime conjecture)**: $\{p, p + 2\}$ であるようなペアは無限に存在する。ただし、 p と $p + 2$ は素数である。
- **ルジャンドル予想 TODO (Legendre's conjecture)**: n^2 と $(n + 1)^2$ の間には必ず素数が存在する。ただし、 n は正の整数である。

基本的なアルゴリズム - Basic algorithms

素数でない n は $a \cdot b$ という積で表すことができるので $a \leq \sqrt{n}$ か $b \leq \sqrt{n}$ となり、2 以上 $\lfloor \sqrt{n} \rfloor$ 以下の因数を持ちます。この性質を利用すれば、ある数が素数であるかどうかの判定およびある数の素因数分解を $O(\sqrt{n})$ 時間で求められます。

次の関数 `prime` は、与えられた数 n が素数かどうかを判定します。この関数は n を 2 から $\lfloor \sqrt{n} \rfloor$ の数で割ろうとし、どれでも割り切れなければ n は素数だとします。

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

次に示す関数 `factors` は n の素因数分解を行います。これは数を順に割れるか判定していき、割れた数を `vector` に追加していきます。この関数は n が 2 から $\lfloor \sqrt{n} \rfloor$ の因数を持たないようにします。この処理が終わったのちに $n > 1$ であるならば、それは最後の素因数です。

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

この関数は、その素因数で数を割った回数だけ `vector` に現れます。例えば、24 について $24 = 2^3 \cdot 3$ なので、これを素因数分解した結果は `[2,2,2,3]` となります。

エラトステネスの篩 - Sieve of Eratosthenes

エラトステネスの篩 (ふるい)(**sieve of Eratosthenes**) とは $2 \dots n$ の間の数が素数かどうかチェックし、素数でない場合はその数の素因数を見つけることができる配列を構築する効率的な前処理アルゴリズムです。

このアルゴリズムは、インデックス $2, 3, \dots, n$ を持つ配列の **篩** を構築します。`sieve[k] = 0` であるとき k が素であることを意味し、`sieve[k] ≠ 0` であるときは k が素でなく、素因数の 1 つが `sieve[k]` であることを意味します。

このアルゴリズムは $2 \dots n$ を順番にを 1 回ずつみていきます。新しい素数 x が見つかったと、 x の倍数 ($2x, 3x, 4x, \dots$) は x で割れてしまうため素数ではないとマークします。

例えば $n = 20$ の場合は次のような配列がつけられます。

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

以下のコードでは、篩の各要素は最初はゼロであると仮定した実装です。

```
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
```

```

for (int u = 2*x; u <= n; u += x) {
    sieve[u] = x;
}
}

```

内側のループの計算量は x を回す際には n/x となります。このアルゴリズムの実行時間は以下の通りで $O(n \log n)$ に非常に近い時間計算量であることが示されます。

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

このアルゴリズムは、数 x が素数の場合にのみ内側ループが実行されるのでより効率的に動作します。このアルゴリズムの実行時間は $O(n \log \log n)$ で、 $O(n)$ に近い動作をします。

ユークリッドの互除法 - Euclid's algorithm

a と b の最大公約数 - **GCD(greatest common divisor)** は a, b を割り切れる最大の数で、 $\gcd(a, b)$ と表現します。また、同様に**最小公倍数 - LCM(least common multiple)** は a, b で割り切れる最小の数で、 $\text{lcm}(a, b)$ で表現します。例として、 $\gcd(24, 36) = 12$ であり $\text{lcm}(24, 36) = 72$ です。

この2つの関係を示します。

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

さて、ユークリッドの互除法- **Euclid's algorithm**^{*1} は2つの数間の最大公約数を求める効率的な手法です。このアルゴリズムは以下に基づきます。

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

例えば、以下の通りになります。

$$\gcd(24, 36) = \gcd(36, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

このアルゴリズムは以下のように実装されます。

```

int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}

```

^{*1} Euclid was a Greek mathematician who lived in about 300 BC. This is perhaps the first known algorithm in history.

ユークリッドの互除法は、 $n = \min(a, b)$ のとき、 $O(\log n)$ の時間で動きます。最悪のケースは a と b が連続したフィボナッチ数の時です。例を示します。

$$\gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1.$$

オイラー関数, Euler's totient function

$\gcd(a, b) = 1$ であるとき、 a と b は互いに素です。オイラー関数 (**Euler's totient function**), 別名オイラーの ϕ 関数やオイラーのトーシェント関数とも呼ばれるは $\varphi(n)$ で示され、1 から n の n と素な整数の個数です。例えば $\varphi(12) = 4$ でこれは、1, 5, 7, 11 の 4 つが 12 と素であるためです。

$\varphi(n)$ の値は n を素因数分解した結果から計算できます。

$$\varphi(n) = \prod_{i=1}^k p_i^{\alpha_i-1} (p_i - 1).$$

例えば $\varphi(12) = 2^1 \cdot (2-1) \cdot 3^0 \cdot (3-1) = 4$ です。 n が素数なら $\varphi(n) = n - 1$ となります。

21.2 mod の計算 - Modular arithmetic

mod の計算 (modular arithmetic) では、ある定数 m に対して数が $0, 1, 2, \dots, m-1$ だけを使うようにします。例えば $m = 17$ のとき、75 は $75 \bmod 17 = 7$ となります。いくつか代表的な式を表します。

$$\begin{aligned} (x+y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\ (x-y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\ (x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\ x^n \bmod m &= (x \bmod m)^n \bmod m \end{aligned}$$

累乗の mod - Modular exponentiation

$x^n \bmod m$ は以下のように $O(\log n)$ で高速に求めることができます。

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ が偶数} \\ x^{n-1} \cdot x & n \text{ が奇数} \end{cases}$$

ここで重要なのは、偶数 n の場合に $x^{n/2}$ の値を一度だけ計算していることです。 n が偶数の場合に n は常に半分になるので $O(\log n)$ となります。 $x^n \bmod m$ の実装例を示します。


```

int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x, n/2, m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}

```

フェルマーの定理とオイラーの定理 - Fermat's theorem and Euler's theorem

フェルマーの定理は、 m が素数かつ x と m が互いに素である時、

$$x^{m-1} \bmod m = 1$$

であり、さらに以下の通りとなります。

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

オイラーの定理 - **Euler's theorem** は、 x と m が素であるとき、以下のように示されます。

$$x^{\varphi(m)} \bmod m = 1$$

オイラーの定理では、 m が素数である時、 $\varphi(m) = m - 1$ であるため、フェルマーの定理が導かれます。

mod での逆数 - Modular inverse

モジュロ m における x の逆数である x^{-1} は以下のようなものです。

$$xx^{-1} \bmod m = 1.$$

例えば $x = 6$ で $m = 17$ であるとき、 $6 \cdot 3 \bmod 17 = 1$ であり、 $x^{-1} = 3$ となります。

mod の逆数とは、mod の状態における除算相当であるので、 x^{-1} をかけると m の mod 下で x で割った値を求めることができます。例えば $36/6 \bmod 17$ というのは、 $36 \bmod 17 = 2$ かつ $6^{-1} \bmod 17 = 3$ なので、 $2 \cdot 3 \bmod 17$ として求めることができる。

mod の逆数とは常に存在するわけではないことに注意してください。例えば $x = 2$ 、 $m = 4$ とすると

$$xx^{-1} \bmod m = 1$$

となり正しくありません。2 の倍数はすべて偶数なので、 $m = 4$ のときあまりは 1 とはなりません。 $x^{-1} \bmod m$ は、 x と m が共に素のときだけ存在することに気をつけてください。

もし、 \bmod の逆元が存在する時には次の式が成立します。

$$x^{-1} = x^{\varphi(m)-1}.$$

もし、 m が素数であるなら次のとおりです。

$$x^{-1} = x^{m-2}.$$

例を示します。

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

\bmod の累乗でみたアルゴリズムを用いてモジュロの逆数は効率的に計算することができます。これにはオイラーの定理を利用します。まず、モジュロの逆数は次の式を満たす必要があります。

$$xx^{-1} \bmod m = 1.$$

一方、オイラーの定理によれば、

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

このため、 x^{-1} と $x^{\varphi(m)-1}$ は等しいことがわかります。

コンピュータ上での演算について - Computer arithmetic

プログラミング上では、 k をデータ型のビット数とすると、符号なし整数とは 2^k で表現されます。これは、数値が大きくなりすぎると、折り返されてしまうのが普通です。

例えば、C++ において、unsigned int というのは、モジュロ 2^{32} で表現されます。例えば次のコードは unsigned int の 123456789 である変数を定義します。この数を二乗すると以下ようになります。 $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

21.3 方程式を解く - Solving equations

ディオファントス方程式 - Diophantine equations

ディオファントス方程式 - **Diophantine equation** は次のような方程式のことです。

$$ax + by = c,$$

a, b, c は定数で x と y を求めたいとします。また、これらは全て整数とします。例えば、 $5x + 2y = 11$ の時、 $x = 3, y = -2$ です。

この方程式はユークリッドの互除法を活用することで効果的に解くことができます。ユークリッドの互除法を拡張することで x, y を次のようにみつけることができます。

$$ax + by = \gcd(a, b)$$

この方程式は、 c が $\gcd(a, b)$ で割り切れる場合に解が存在し、そうでなければ解は存在しません。

例えば次の方程式を考えます。

$$39x + 15y = 12$$

$\gcd(39, 15) = 3$ で $3 \mid 12$ であるため、この式は解けます。さて、ユークリッドのアルゴリズムが 39 と 15 の最大公約数を計算するとき、次のような関数呼び出しが行われます。

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

これは次のような式ということができます。

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

これにより、次のようになります。

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

そして、それぞれを 4 倍すると、

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

ということから、 $x = 8, y = -20$ が導けました。

尚、ディオファントス方程式の解は一意ではないことに注意してください。一つの解がわかれば無限に解を作ることができます。ある組 (x, y) が解であるとき全ての解のペアは次に示すように求められます。

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

ここで k は任意の整数です。

中国人余剰定理 - Chinese remainder theorem

中国人余剰定理 (Chinese remainder theorem) は次のような式を解きます。

$$\begin{aligned} x &= a_1 \bmod m_1 \\ x &= a_2 \bmod m_2 \\ \dots \\ x &= a_n \bmod m_n \end{aligned}$$

m_1, m_2, \dots, m_n は互いに素だとします。

x_m^{-1} はモジュロ m における x の逆数とします。

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

この表記法を用いると、方程式の解は以下のようになります。

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}.$$

各 $k = 1, 2, \dots, n$ に対して、

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

となり、なぜなら

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

であるからです。和の他の項はすべて m_k で割り切れるので、余りには影響しないため、 $x \bmod m_k = a_k$ となります。

例えば以下の式を考えましょう。

$$\begin{aligned} x &= 3 \bmod 5 \\ x &= 4 \bmod 7 \\ x &= 2 \bmod 3 \end{aligned}$$

以下のようになります。

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

一旦、解 x が見つければ、他の解を無限に見つかります。なぜなら、以下の形の全てが解となるためです。

$$x + m_1 m_2 \cdots m_n$$

21.4 その他 - Other results

Lagrange's theorem

楽ランジュの定理 (Lagrange's theorem) とは、すべての正の整数が 4 つの二乗の和で表せるというもので、例えば、123 は以下のように示す。

ゼッケンドルフの定理 - Zeckendorf's theorem

ゼッケンドルフの定理 (Zeckendorf's theorem) は、すべての正の整数がフィボナッチ数の和として一意に表現され、数字が等しいフィボナッチ数あるいは連続したフィボナッチ数にはならない、という定理です。例えば、74 は $55 + 13 + 5 + 1$ です。

TODO: Pythagorean triples

ピタゴラスの定理 (Pythagorean triple) は、 $a^2 + b^2 = c^2$ を満たす (a, b, c) の辺を持つ三角形は直角三角形であるというものです。例えば、 $(3, 4, 5)$ はこれを満たす組です。

(a, b, c) がピタゴラスの定理を満たす組であるとき、 (ka, kb, kc) もピタゴラスの定理を満たす組です。これらの組は a, b, c が共に素であれば**原始項 (primitive)** と呼ばれ、乗数 k を用いて a, b, c の組を原始項から作ることができます。ここで、**ユークリッドの公式**を使えば、すべてのピタゴラスの定理を満たす原始項を作り出すことができます。

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

ここで、 $0 < m < n$ であり n と m は互いに素です。また n か m の最低 1 つは偶数である必要があります。 $m = 1, n = 2$ の時、最小のピタゴラスの定理の組が作られます。

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

ウィルソンの定理 - Wilson's theorem

ウィルソンの定理 (Wilson's theorem) は n が素数のとき以下が成り立つというものです。

$$(n - 1)! \bmod n = n - 1.$$

素数である 11 を例に挙げると、

$$10! \bmod 11 = 10,$$

素数でない 12 を挙げると、

$$11! \bmod 12 = 0 \neq 11.$$

ウィルソンの定理は、ある数が素数であるかどうかを調べるために用いることができますが、 n が大きいときには、 $(n-1)!$ の値を計算することは難しいので実際に素数を求めるために使うのは困難です。

第 22 章

組合わせ論 - Combinatorics

組合わせ論 - Combinatorics は組合わせを数える方法を研究する分野です。通常、これらは各組合わせを個別に数え上げるのではなく効率的に組み合わせを数える方法を見つけます。

例として、ある整数 n を正の整数の和として表現する方法の数を数える問題を考えます。例えば 4 には 8 通りの表現があります。

- | | |
|-------------|---------|
| • $1+1+1+1$ | • $2+2$ |
| • $1+1+2$ | • $3+1$ |
| • $1+2+1$ | • $1+3$ |
| • $2+1+1$ | • 4 |

組合せ問題では再帰関数がよく使われます。この問題では n を表現する数を示す関数 $f(n)$ を定義してみましょう。例えば、上記の例では $f(4)=8$ です。この関数の値は以下のように再帰的に計算することができます。

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \dots + f(n-1) & n > 0 \end{cases}$$

$f(0)=1$ はこれは空集合が数 0 を表すので自明です。次に $n > 0$ のとき、和の第 1 項目の選び方をすべて考えます。第 1 項目を k と定めると和の残りの部分には $f(n-k)$ の表現が存在します。したがって $k < n$ である $f(n-k)$ の形のすべての値の和を計算します。

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \end{aligned}$$

これは一般的な式で示せます。

$$f(n) = 2^{n-1},$$

これは + と - の位置は $n-1$ 通りありそのうちの任意の部分集合を選ぶことができる、という事実に基づいています。(TODO: ここあってる?)

22.1 二項係数 - Binomial coefficients

二項係数 - binomial coefficient は $\binom{n}{k}$ で表されます。これは n 個の要素の集合から k 個の要素の部分集合を選ぶ時の組み合わせの数です。例えば、 $\binom{5}{3} = 10$ で、例えば集合 $\{1, 2, 3, 4, 5\}$ を考えた時に 3 つとるという組み合わせは次の 10 通りあります。

$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$

公式 1 - Formula 1

二項係数は次のように再起的に計算できます。

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

これは集合の中のある要素 x を固定するアイデアと言えます。 x が部分集合に含まれる場合は、 $n-1$ 個の要素から $k-1$ 個の要素を選ぶことになり、 x が部分集合に含まれない場合は、 $n-1$ 個の要素から k 個の要素を選ぶことになるためです。次の点に注意してください。

$$\binom{n}{0} = \binom{n}{n} = 1,$$

空の部分集合とすべての要素を含む部分集合を構成する方法は常に 1 つしかありません。

公式 2 - Formula 2

また次が成り立ちます。

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

n 個の要素の並べ換えは $n!$ 個です。すべての順列を調べ、常に順列の最初の k 個の要素を部分集合に含めましょう。部分集合の中の要素と部分集合の外の要素の順序は重要ではないので、 $k!$ と $(n-k)!$ で割れば良いです。

補足 - Properties

$$\binom{n}{k} = \binom{n}{n-k},$$

となります。 n 個の要素からなる集合を 2 つの部分集合に分割して 1 つ目の部分集合には k 個の要素を含め、2 番目は $n-k$ 個の要素を含む場合を考えればこれは自明です。

二項係数の和は

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

で示されます。二項係数という名前の由来は $(a+b)$ を n 乗したときにわかります。次の通りです。

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1 b^{n-1} + \binom{n}{n}a^0 b^n.$$

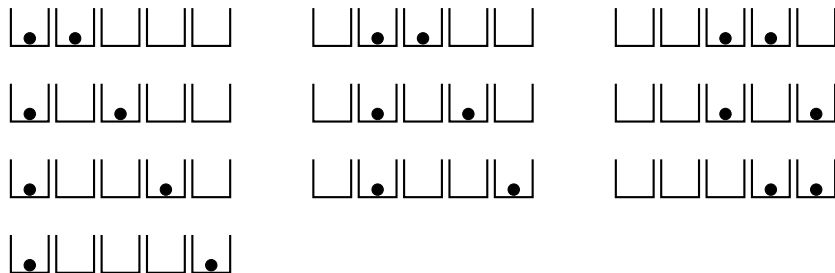
二項係数はパスカルの三角形 - **Pascal's triangle** と密接な関係があります。

$$\begin{array}{ccccccc} & & & & 1 & & & & \\ & & & & 1 & & 1 & & \\ & & & 1 & & 2 & & 1 & \\ & & 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{array}$$

箱とボールの問題 - Boxes and balls

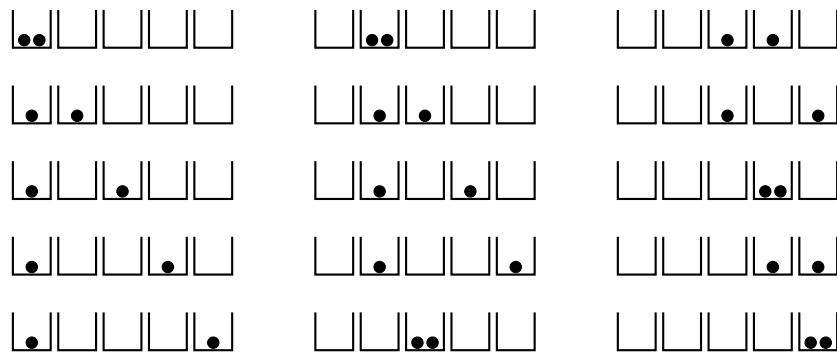
”箱とボールの問題 - Boxes and balls” は二項係数を考える上で非常に有名な問題です。 k 個の玉を n 個に入れる方法を数えます。3 つのシナリオを考えてみましょう。

シナリオ 1: 各ボックスには、最大で 1 個のボールを入れることができます。例えば、 $n=5$ 、 $k=2$ のとき、解は 10 です。



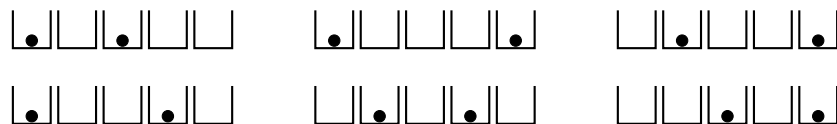
このシナリオでは答えは二項係数そのままです。 $\binom{n}{k}$ となります。

シナリオ 2: 箱には複数の球を入れることができる。例えば、 $n = 5$ で $k = 2$ の場合、解は 15 です。



ボールを箱に入れる作業を記号 "o" と "→" で表現しましょう。はじめに一番左の箱にいるとします。記号 "o" は現在の箱に玉を入れることを意味し、記号 "→" は次の右隣の箱に移動することを意味します。この表記法を用いると、各解は "o" という記号を k 回、"→" という記号を $n-1$ 回含む文字列となります。例えば、上の図の右上の解は、"→ → o → o →" という文字列に相当します。そのため答えは $\binom{k+n-1}{k}$ です。

シナリオ 3: 各箱にボールを入れる。しかし隣合う箱にボールが入ってはいけません。 $n = 5, k = 2$ とするとき、次の 6 通りが考えられます。



このシナリオでは、 k 個のボールが箱に入れられ、隣合う 2 つの箱の間に空の箱があるとしましょう。課題は、残りの空き箱の位置を選択することである。このような箱は $n-2k+1$ 個あり、その位置は $k+1$ が考えられます。

ここでシナリオ 2 の式を思い出すと $\binom{n-k+1}{n-2k+1}$ であることがわかります。

多項係数 - Multinomial coefficients

多項係数 - multinomial coefficient は、

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \cdots k_m!},$$

であり、 n 個の要素をサイズ k_1, k_2, \dots, k_m の部分集合に分割できる方法の数に等しくなります。この時、 $k_1 + k_2 + \cdots + k_m = n$ とします。

多項係数は二項係数の一般化と見ることができ、 $m = 2$ なら上式は二項係数の式となります。

22.2 カタラン数 - Catalan numbers

カタラン数 - Catalan number C_n は、 n 個の左括弧と n 個の右括弧からなる有効な括弧式の数に等しいような数字です。例えば $C_3 = 5$ ですが、左右 3 つの括弧を使って次のような括弧式が作れます。

- $()()()$
- $((()))$
- $()(())$
- $((()))$
- $(())()$

括弧表現 - Parenthesis expressions

有効な括弧表現は次の定義で示されます。

- 空の括弧は有効です
- 式 A が有効であるとき、 (A) は有効です
- A と B が有効である時、 AB は有効です

有効な括弧式のもう 1 つの特徴は、このような式の任意の接頭辞を選ぶと、そこに含まれる右括弧以上の左括弧が含まれていなければならないことです。また、完全な式には、左括弧と右括弧が同じ数だけ含まれていなければなりません。

公式 1: Formula 1

カタラン数は次の式で求めることができます。

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

この和は式を 2 つの部分に分割して両方の部分が有効な式であり最初の部分ができるだけ短い空でないものの数を調べます。任意の i について最初の部分は $i+1$ 組の括弧を含むため式の数値は次の値の積となります。

- C_i : 一番外側の括弧を除いた最初の部分の括弧を使った式の組み立て方の数
- C_{n-i-1} : 2 番目の括弧を使った式の組み立て方の数

なお、 $C_0 = 1$ となります。これは、ゼロ個の括弧の組を使って空の括弧式を構成することができるためです。

Formula 2

カタラン数は二項係数から求めることもできます。

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

n 個の左括弧と n 個の右括弧を含む有効とは限らない括弧式を構成する数は全部で $\binom{2n}{n}$ です。これかから有効でないものの数を計算します。

括弧式が有効でない場合というのは、ある地点に置いて右括弧の数が左括弧の数を上回るような接頭辞でなければいけません。そのような接頭辞に属するそれぞれの括弧を逆にしてみます。例えば、 $()()()$ は接頭辞 $()$ を含んでおり、接頭辞を反転させると $)((()$ となります。

これを行うと式は $n+1$ 個の左括弧と $n-1$ 個の右括弧で構成されます。このような $\binom{2n}{n+1}$ の式は有効でない括弧式の数となります。したがって、有効な括弧式の数
は、次の式で計算できます。

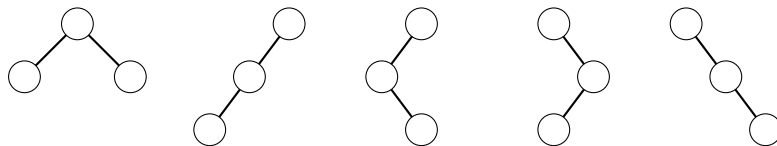
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

木の数え上げ - Counting trees

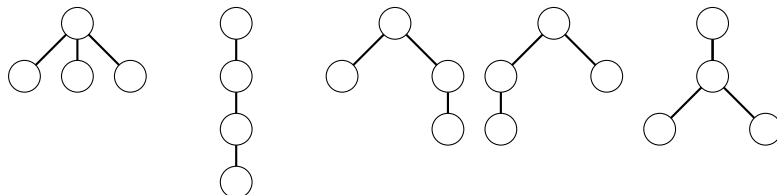
カタラン数は木とも関係があります。

- n 個のノードを持つ二分木の数は C_n
- n 個のノードを持つ根付き木の数は C_{n-1}

$C_3 = 5$ です。さて二分木は次のようになります。



根付き木は次のようになります。

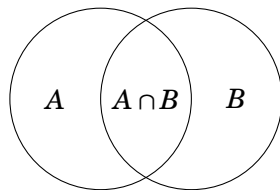


22.3 包除原理 - Inclusion-exclusion

包除原理 - Inclusion-exclusion とは、集合の重なる部分の大きさが分かっているときに、その集合の和の大きさを数えるテクニックです。2つの集合の例としては、

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

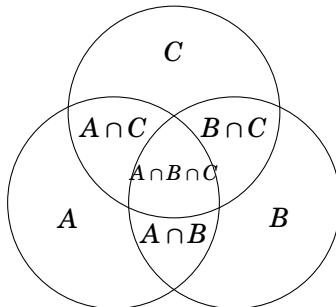
となって A と B は集合であり $|X|$ という表現は X の集合の大きさです。これを図に示します。



図のようにまず A と B の面積から $A \cap B$ の面積を差し引けば、 $A \cup B$ の面積を計算することができます。

この考え方は集合の数が多くなっても同じ考え方ができます。3つの集合の考え方と図を示します。

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$



一般的に $X_1 \cup X_2 \cup \dots \cup X_n$ のサイズの大きさは集合 X_1, X_2, \dots, X_n の全ての重なる部分を調べることによって計算することができます。この時に重なる部分の集合の数の偶奇に注目すると良いです。交差点が奇数の集合を含む場合は加算で、偶数の集合を含む場合は減算します。

重なる部分のサイズを計算する式は以下ようになります。

$$|A \cap B| = |A| + |B| - |A \cup B|$$

集合が3つの場合は次のようになります。

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

完全順列 - Derangements

完全順列 - derangements とは、 n 個の要素 $\{1, 2, \dots, n\}$ の並び替えのうちで元の位置と一致する要素がないような順列のことです。 $n = 3$ とすると 2 つの完全順列があり: $(2, 3, 1)$ と $(3, 1, 2)$ です。

これを求める一つのアプローチは包除原理を用いることができます。 X_k を位置 k に要素 k を含む並び換えの集合としましょう。例えば、 $n = 3$ のときに集合は次のようになります。

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

これらを用いると、完全順列は以下に等しくなります。

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

この集合和の大きさを計算すれば十分になります。包除原理を用いると、これは交点の大きさを計算すればよく、効率的に計算できます。例えば、 $n = 3$ $|X_1 \cup X_2 \cup X_3|$ のサイズは以下のようにになります。

$$\begin{aligned} &|X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

$3! - 4 = 2$. とわかりました。

尚、この問題は包除原理を使わずに解けることがわかっています。 $f(n)$ を $\{1, 2, \dots, n\}$ に対応する完全順列の数とします。この時、以下のように示せます。

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

この式は完全順列の中で要素 1 がどのように変化するかのパターンを考えることで導き出すことができます。要素 1 に代わる要素 x の選び方は $n-1$ 通りです。それぞれの選択において、2 つの選択肢がある。

Option 1: 要素 x を要素 1 に置き換える。この後、残る課題は、 $n-2$ 個の要素からなる完全順列を構成することである。

Option 2: 要素 x を 1 以外の要素に置き換える。要素 x を要素 1 に置き換えることはしないため、 $n-1$ 個の要素の完全順列を構成しなければならず、他の要素もすべて変更しなければならならぬ。

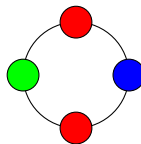
22.4 バーンサイドの補題 - Burnside's lemma

バーンサイドの補題 - Burnside's lemma は、対称的な組み合わせのグループごとに重複がないように組み合わせの数を数えるために利用します。バーンサイドの補題は組み合わせの数が、

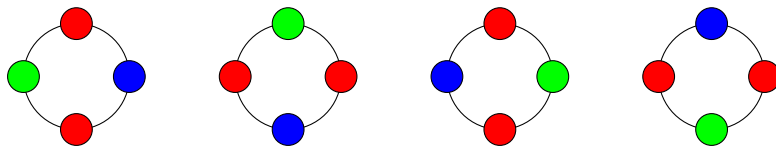
$$\sum_{k=1}^n \frac{c(k)}{n},$$

で示され、組み合わせの位置を変える方法が n 通りあり、そこに k 番目の方法を適用しても変化しない $c(k)$ の組み合わせがあるとします。

真珠の色が m 色あるとして n 個の真珠を使ったネックレスの組み合わせ数を計算してみましょう。2つのネックレスが回転して似ている場合、対称（同一の意味）であると言えます。



に対して、以下のネックレスは同一であるとしましょう。



ネックレスを時計回りに $0, 1, \dots, n-1$ ステップ回転させることができるので、ネックレスの位置を変える方法は n 通りあります。ステップ数が 0 の場合、 m^n のネックレスはすべて同じで、ステップ数が 1 の場合、各真珠の色が同じである m 個のネックレスだけが同じままである。

より一般的には、ステップ数を k としたとき、合計で

$$m^{\gcd(k,n)}$$

のネックレスが同じになります。 $\gcd(k,n)$ は k と n の最大公約数です。この理由は、 $\gcd(k,n)$ の大きさの真珠のブロックは互いに置き換えられるためです。したがって、バーンサイドの補題によれば、首飾りの個数は

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

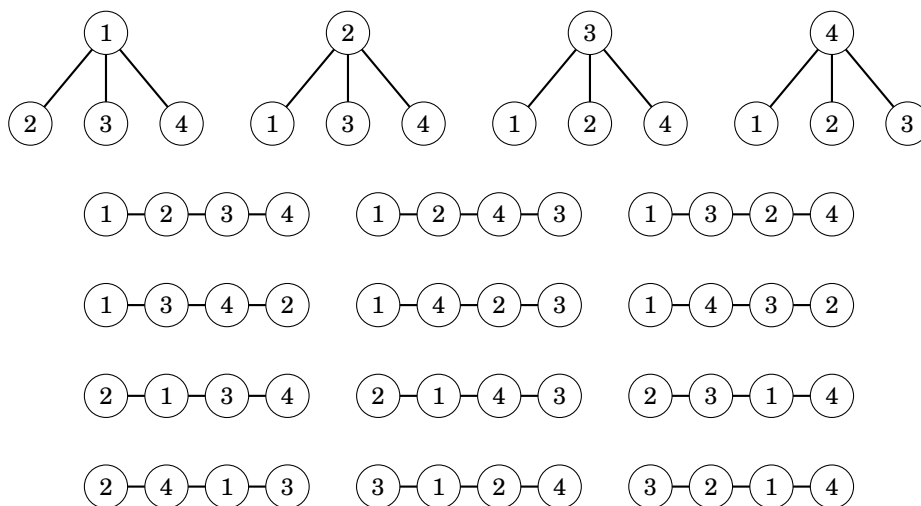
で示されて長さ 4 で色が 3 なら以下の通りとなります。

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 ケイリーの公式 - Cayley's formula

ケイリーの公式 - Cayley's formula は、 n 個のノードを含む n^{n-2} のラベル付き木が存在することを述べています。ノードは $1, 2, \dots, n$ とラベル付けされ、2つの木はその構造かラベルが異なれば異なる木としましょう。

例えば、 $n=4$ のとき、ラベル付き木の本数は $4^{4-2} = 16$ となる。

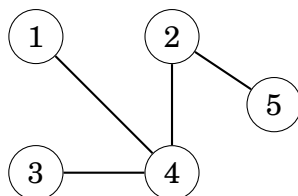


次に、プリューファーコードを使ってケイリーの公式がどのように導かれるのかを見ていきます。

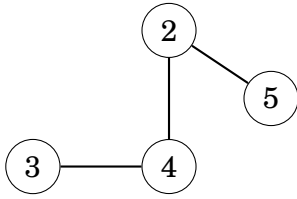
プリューファー列 - Prüfer code

プリューファー列 - Prüfer code とはラベル付きツリーを記述する $n-2$ 個の数値です。この数値は木から $n-2$ 枚の葉を除去する過程を経て構成されます。各ステップで最も小さいラベルを持つ葉が取り除かれ、その唯一の隣接葉のラベルをコードに追加します。

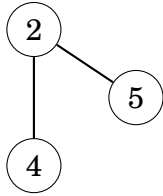
次のようなグラフのプリューファーコードを計算してみましょう。



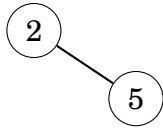
まず、ノード 1 を削除してノード 4 をコードに追加します。



そして、ノード 3 を削除しノード 4 を追加するコードになります。



最後に、ノード 4 を削除しノード 2 をコードに追加します。



このグラフのプリューファー列は [4,4,2] です。

どのような木に対してもプリューファー列を構成することができます。さらに重要なことはプリューファー列から元の木を再構成することができることです。したがってノード数 n のラベル付き木の数は、サイズ n のプリューファー列の数 n^{n-2} に等しくなります。

第 23 章

行列 - Matrices

行列 (matrix) はプログラミングにおける 2 次元配列に相当する数学の用語です。

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

はサイズ 3×4 の行列であり 3 行 4 列の行列と呼ばれます。[i, j] という表記がよく用いられ行列の i 行 j 列を意味します。上の行列では $A[2,3] = 8$ and $A[3,1] = 9$ です。行列の特殊な例として大きさが 1 次元の行列である **ベクトル** があります。つまり $n \times 1$ の行列のことで以下のようになります。

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

これは 3 つの要素からなるベクトルです。

転置行列 - transpose A^T は A の行と列が入れ替わったものです。つまり、 $A^T[i,j] = A[j,i]$ になります。例をみてみましょう。

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

正方行列 - square matrix 行と列が同じ数の行列は正方行列と呼ばれます。例を示します。

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

23.1 行列と行列の演算 - Operations

行列 A と B の和 $A+B$ は、行列が同じ大きさである場合に操作できます。 A と B の同じ行と列の要素の和が答えとなります。

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

行列 A に対する x での乗算は各要素に x をかけたものとなります。

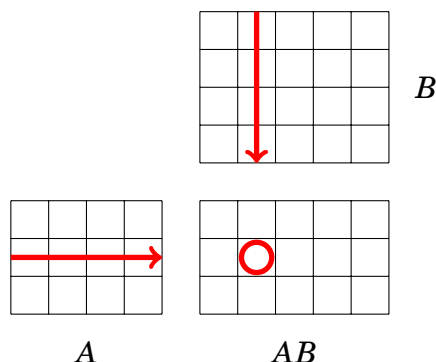
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

行列同士の乗算 - Matrix multiplication

行列 A と B の積 AB は、 A がサイズ $a \times n$ であり、 B がサイズ $n \times b$ の時に定義されます。言い換えれば、 A の幅が B の高さに等しい時、と言えます。

$$AB[i,j] = \sum_{k=1}^n A[i,k] \cdot B[k,j].$$

AB の各要素は、 A の要素の積の和であるという考え方をします。



$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

行列の乗算において $A(BC) = (AB)C$ は成立します。ただし、可換ではないので $AB = BA$ は成立するとは限りません。

単位行列 - identity matrix とは、対角線上の各要素が 1 で、それ以外の要素を 0 とする正方行列です。例えば、次の行列は、 3×3 の単位行列です

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

行列に単位行列をかけても行列は変わりません。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{であり} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

基本的なアルゴリズムを用いると2つの $n \times n$ の行列の積は $O(n^3)$ で計算できます。行列の乗算にはより効率的なアルゴリズムがあります。それは数学的なものであり競技プログラミングでは使われません。^{*1}

行列の累乗 - Matrix power

A^k は A が正方行列の時に定義されます。これは次のようになります。

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ times}}$$

例をあげます。

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

また、 A^0 は単位行列とします。

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

A^k は 21.2 のアルゴリズムを使えば $O(n^3 \log k)$ で求められます。

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

行列式 - Determinant

行列式 - determinant は A が正方行列である場合に定義されます。 A が 1×1 の大きさであれば $\det(A) = A[1, 1]$ となります。より大きな行列の行列式は次の式を用いて再帰的に計算されますこの時にが用いられます

$$\det(A) = \sum_{j=1}^n A[1, j] C[1, j],$$

$C[i, j]$ は A における $[i, j]$ の余因子で次の様に示されます。

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

^{*1} 最も最初のこの工夫は 1969 年に発表された Strassen のアルゴリズム [63] で、 $O(n^{2.80735})$ です。現在の最も優れたアルゴリズム [27] は $O(n^{2.37286})$ です。

$M[i, j]$ は A から i 行 j 列を削除して得られる行列です。係数 $(-1)^{i+j}$ により符号は毎回入れ替わります。

$$\det\begin{pmatrix} 3 & 4 \\ 1 & 6 \end{pmatrix} = 3 \cdot 6 - 4 \cdot 1 = 14$$

であり、

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

A の行列式は $A \cdot A^{-1} = I$ (I は単位行列) となる逆行列 A^{-1} が存在するかどうかを教えてください。 A^{-1} は $\det(A) \neq 0$ のときに正確に存在して次の式で計算できます。

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

$$\underbrace{\begin{bmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{bmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{bmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{bmatrix}}_{A^{-1}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_I.$$

23.2 線型回帰 - Linear recurrences

線型回帰 - linear recurrence は $f(n)$ に対して $f(0), f(1), \dots, f(k-1)$ などの初期値が与えられている時に、大きな n である

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k),$$

を求めたいとします。ここで c_1, c_2, \dots, c_k は定数です。

動的計画法では、 $f(0), f(1), \dots, f(n)$ を順次計算することで、任意の $f(n)$ を $O(kn)$ で計算できる。ですが、 k が小さい場合には行列演算を用いると、 $O(k^3 \log n)$ で求められ効率的なことがあります。

フィボナッチ数 - Fibonacci numbers

線型回帰としてよくあるのはフィボナッチ数です。

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

$k=2$ の時、 $c_1 = c_2 = 1$ です。

フィボナッチ数の計算を効率的に行うために、フィボナッチ式をサイズ 2×2 の正方行列 X で表現すると、次のようになります。

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

$f(i)$ と $f(i+1)$ は X に対する”入力”で X から $f(i+1)$ と $f(i+2)$ を求めていきます。

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

例としては、

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

こうして $f(n)$ を計算できます。

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

X^n の値は $O(\log n)$ 時間で計算できるため、 $f(n)$ の値も $O(\log n)$ で計算できます。

一般化 - General case

では $f(n)$ が任意の線形漸化式である一般的なケースを考えていきます。目的は以下のような行列 X を構成することです。

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

この様な行列は次のようになります。

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

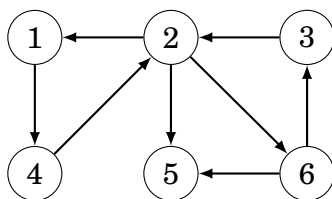
最初の $k-1$ 行は、1つの要素が1であることを除いて、各要素が0です。これらの行は、 $f(i)$ を $f(i+1)$ 、 $f(i+1)$ を $f(i+2)$ と置き換えていきます。最後の行には新しい値 $f(i+k)$ を計算するための漸化式の係数が含まれます。さて、 $f(n)$ は次の式を用いて $O(k^3 \log n)$ で計算できます。

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

23.3 グラフと行列 - Graphs and matrices

パスのカウント - Counting paths

グラフの隣接行列の累乗には興味深い性質があります。 V が重みなしグラフの隣接行列であるとき、行列 V^n はグラフのノード間の n 本のエッジのパスの数を含んでいます。



この場合の隣接行列は、

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

ここで、次を考えます。

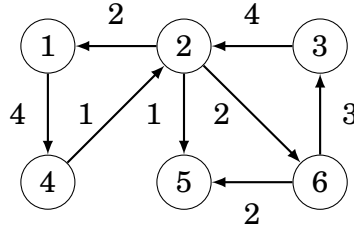
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

これはパスが4のノードを示します。例えば、 $V^4[2,5] = 2$ ですが、これはノード2とノード5の長さ4のパスは2つであることを示します。 $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ と $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$ です。

最短経路 - Shortest paths

重み付きグラフも同様の考え方で、ノードのペアごとに、その間にちょうど n 本のエッジを含むパスの最小の長さを計算することができます。これを計算するためにはパスの数を計算するのではなく、パスの長さを最小にするように行列の乗算を新たに定義する必要があります。

以下の例を考えます。



∞ は辺が存在しないとし、各パスの重さを隣接グラフで表現します。

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

ここで、

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j]$$

という乗算を以下の様に定義します。

$$AB[i, j] = \min_{k=1}^n A[i, k] + B[k, j]$$

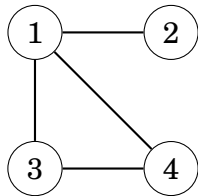
の代わりに最小値を、積の代わりに要素の和を計算します。これで行列の累乗はグラフの最短経路に対応します。

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

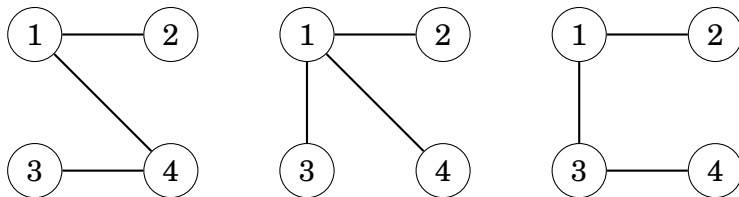
ノード 2 からノード 5 までの 4 辺のパスの最小長は 8 です。このようなパスは、 $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$.

キルヒホッフの定理 - Kirchhoff's theorem

キルヒホッフの定理 - **Kirchhoff's theorem** はグラフの全域木の数を実数行列の行列式として計算する方法です。



これは 3 つの全域木があります。



全域木の数进行計算するために、 $L[i,i]$ をノード i の次数とし、ノード i と j の間にエッジがあれば $L[i,j] = -1$ 、なければ $L[i,j] = 0$ のラプラシアン行列 - **Laplacian matrix** を構成します。上記のグラフのラプラシアン行列は次のようになります。

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

全域木の数、は、 L から任意の行と列を削除したときに得られる行列の行列式に等しいです。

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3.$$

L からどの行と列を削除しても、行列式は常に同じです。

なお、22.5 章の Cayley の公式は Kirchhoff の定理の特殊な例で、ノード数 n の完全グラフです。

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

第 24 章

確率 - Probability

確率 - probability は 0 から 1 の間の実数で表されるある事象がどの程度の確率で起こるかを示すものです。ある事象が確実に起こるなら確率は 1 であり、確実にあり得ないならその確率は 0 です。ある事象の確率は $P(\dots)$ と表記され、3 つの点はその事象を表します。

サイコロを投げるとき、その結果は 1 から 6 の整数であり、それぞれの結果が出る確率は $1/6$ です。このため、次のような確率が計算できる。

- $P(\text{"4 がでる"}) = 1/6$
- $P(\text{"6 以外"}) = 5/6$
- $P(\text{"偶数"}) = 1/2$

24.1 確率の計算 - Calculation

ある確率を計算するには組合せ論を用いるか、その事象が発生する過程をシミュレートする方法があります。シャッフルした山札から同じ数字のカードを 3 枚引く確率を計算してみます。(例えば、♠8, ♣8, ◇8)。

方法 1

次の式を使うことができます。

$$\frac{\text{望ましい数量}}{\text{あり得る全体の数}}$$

この問題では、各カードの価値は同じです。このような結果は、 $13\binom{4}{3}$ あります。ある数を引く可能性は 13 パターンがあり、その中で 4 つのスーツのうちから 3 つを引くのは $\binom{4}{3}$ のパターンがあります。ここで全体の引くパターンは 52 枚のカー

ドから 3 枚のカードを選ぶので、 $\binom{52}{3}$ です。この確率は以下の様になります。

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

方法 2

今度はプロセスをシミュレーションするアプローチを考えます。この例では、3 枚のカードを引くので、3 回のプロセスを実行します。まず、1 枚目のカードは何を選んで良いです。第 2 段階は 51 枚のカードが残っていて、そのうち 3 枚が最初のカードと同じ数なので $3/51$ の確率で成功です。3 番目のステップも $2/50$ の確率で成功です。つまり、全体の処理が成功する確率は以下の様になります。

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.2 事象 - Events

確率における事象は集合として表現できます。

$$A \subset X$$

X はすべての可能な結果全体で、 A は結果の部分集合です。サイコロを振るとその結果は

$$X = \{1, 2, 3, 4, 5, 6\}$$

となり、“偶数である”という集合は以下の通りです。

$$A = \{2, 4, 6\}$$

ある事象 x には確率 $p(x)$ が割り当てられています。ある事象 A の確率 $P(A)$ は、次ように結果の確率の総和として計算することができます。

$$P(A) = \sum_{x \in A} p(x).$$

サイコロを投げる場合、各結果 x に対して $p(x) = 1/6$ であるから、“偶数である”という事象の確率は以下の通りです。

$$p(2) + p(4) + p(6) = 1/2.$$

X の確率は 1、つまり $P(X) = 1$ です。各事象は集合なので標準的な集合演算で操作できます。

- **補集合 - complement** \bar{A} は” A が起こらない” という意味です。例えばサイコロにおいて、 $A = \{2, 4, 6\}$ の補集合は $\bar{A} = \{1, 3, 5\}$ です。
- **集合和 - union** $A \cup B$ は” A か B が起きる” という意味です。 $A = \{2, 5\}$ と $B = \{4, 5, 6\}$ の和は、 $A \cup B = \{2, 4, 5, 6\}$ です。
- **集合積 - intersection** $A \cap B$ は” A と B が起きる” という意味です。 $A = \{2, 5\}$ と $B = \{4, 5, 6\}$ の集合積は $A \cap B = \{5\}$ です。

補集合 - Complement

補集合 \bar{A} は次の様に計算できます。

$$P(\bar{A}) = 1 - P(A).$$

補集合によって、ある問題の逆を解くことで簡単に解けることがあります。サイコロを 10 回投げたとき、少なくとも 1 回 6 が出る確率は次の様に簡単に求められます。

$$1 - (5/6)^{10}.$$

ここで、 $5/6$ は一投の結果が 6 でない確率です。 $(5/6)^{10}$ は 10 投のうちただの一回も 1 投も 6 でない確率となります。そのため、これの補数が問題の答えとなります。

集合和

$A \cup B$ の確率は以下の様に示します。

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

例えば、サイコロを考えて、

$$A = \text{” 偶数である ”}$$

と

$$B = \text{” 4 未満である ”}$$

であるとき、

$$A \cup B = \text{” 偶数であるか 4 未満である ”},$$

という確率は、

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

となります。事象 A と B が**不連続, 排他的 - disjoint**、すなわち $A \cap B$ が空である場合は事象 $A \cup B$ の確率は、単純に以下の様に示します。

$$P(A \cup B) = P(A) + P(B).$$

条件付き確率 - Conditional probability

条件付き確率 - conditional probability

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

というのは B が起こったと仮定した場合の A の確率です。したがって A の確率を計算するときは B に属する結果だけを考えます。先ほどのセットを使うと、

$$P(A|B) = 1/3,$$

となります。 B の結果は $\{1,2,3\}$ で、そのうちの 1 つは偶数だからです。つまり結果が $1 \dots 3$ となった場合の偶数の結果の確率です。

共通部分 - Intersection

条件付き確率を用いると共通部分である $A \cap B$ の確率は以下の様に求めることができます。

$$P(A \cap B) = P(A)P(B|A).$$

A と B が**独立 - independent**なのは次のときです。

$$P(A|B) = P(A) \quad \text{かつ} \quad P(B|A) = P(B),$$

ということは B が起こっても A の確率は変わらず、その逆も同様ということです。さて、この場合に共通部分の確率は

$$P(A \cap B) = P(A)P(B).$$

です。例えば山札からトランプを引く場合に

$A =$ "スーツがクラブである"

$B =$ "数字が 4 である"

は独立です。このため、

$A \cap B =$ "クラブの 4 である"

という確率は次の様に示します。

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52.$$

24.3 確率変数 - Random variables

ランダムな値 - random variable はランダムに生成される値のことです。例えば、2 回サイコロを投げるときの考えられる確率変数は

$$X = \text{” 出た目の和”}.$$

例えば結果が [4,6](最初に 4 で次に 6 だったという意味) であれば、 X の値は 10 とします。

例えば、2 つのサイコロを投げるときに $P(X = 10) = 3/36$ です。結果の総数は 36 であり合計 10 を得るには [4,6], [5,5], [6,4] という 3 通りの可能性があるからです。

期待値 - Expected value

期待値 - expected value $E[X]$ 確率変数 X の平均値を示します。期待値は、以下の和として計算できます。

$$\sum_x P(X = x)x,$$

x は X でありうる全ての数です。

例えばサイコロを投げるときの期待値は以下の通りです。

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

期待値の面白い性質は**線形性 - linearity** です。 $E[X_1 + X_2 + \dots + X_n]$ は常に $E[X_1] + E[X_2] + \dots + E[X_n]$ と等しいです。この式は、確率変数が互いに依存しあっても成り立ちます。

例えば、2 つのサイコロを投げるとき期待される和は以下の通りです。

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

n 個のボールが n 個の箱にランダムに入れられて空箱の期待個数を計算する問題を考えます。各球は等しい確率でどの箱にも入ります。 $n = 2$ の場合、次の様に考えられます。



このとき、空の箱の数は以下の通りです。

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

一般に 1 つの箱が空の確率は、

$$\left(\frac{n-1}{n}\right)^n,$$

で求められます。その箱にボールが入っていないからです。ここで線形性を利用すると期待値は次のようになります。

$$n \cdot \left(\frac{n-1}{n} \right)^n.$$

分布 - Distributions

分布 - distribution とは X が持ちうる各値の確率を示します。分布は、値 $P(X=x)$ から構成されます。2つのサイコロを投げるとき、その和の分布は次のようになります。

x	2	3	4	5	6	7	8	9	10	11	12
$P(X=x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

一様分布 - uniform distribution では確率変数 X は n 個の値 $a, a+1, \dots, b$ を取る時、各値を取る確率は $1/n$ とします。サイコロをに当てはめれば $a=1, b=6$ で、各値 x に対して $P(X=x)=1/6$ です。この様な一様分布における X の期待値は

$$E[X] = \frac{a+b}{2}.$$

二項分布 - binomial distribution において、 n 回の試行があった時、1回の試行が成功する確率を p とすると、確率変数 X は試行の成功回数を数えることになります。つまり、ある値 x の確率は次のとおりです。

$$P(X=x) = p^x (1-p)^{n-x} \binom{n}{x},$$

ここで p^x と $(1-p)^{n-x}$ はそれぞれ成功と失敗に対応しています。 $\binom{n}{x}$ はその試行が選ばれる回数です。

例えばサイコロを 10 回投げ、6 が 3 回出る確率は、 $(1/6)^3 (5/6)^7 \binom{10}{3}$ です。

二項分布における X の期待値は次の通りです。

$$E[X] = pn.$$

幾何分布 - geometric distribution とは、試行が成功する確率を p とし、最初の成功が起こるまで続けます。確率変数 X は必要な試行回数を数えるもので、ある値 x の確率は以下のとおりです。

$$P(X=x) = (1-p)^{x-1} p,$$

先ほどと同様に $(1-p)^{x-1}$ は失敗した試行に対応し p は最初の成功した試行に対応します。

ここでサイコロを 6 が出るまで振るとして、投げた回数がちょうど 4 回である確率は $(5/6)^3 1/6$ となります。

幾何分布における X の期待値は次の通りです。

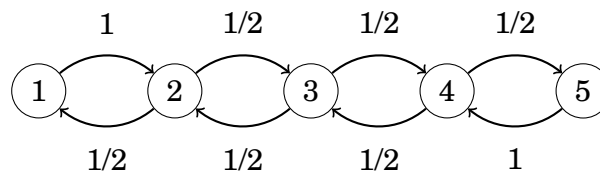
$$E[X] = \frac{1}{p}.$$

24.4 マルコフ連鎖 - Markov chains

マルコフ連鎖 - Markov chain は状態とその間の遷移からなるランダムなプロセスのことです。ここで各状態から他の状態に移行する確率は定義されています。マルコフ連鎖は状態をノードとして遷移をエッジとするグラフで表現することができます。

次のような問題を考えます。最初 n 階建ての建物の 1 階にいます。各ステップにおいてランダムに 1 階上か 1 階下に移動します。ただし、1 階からは上にしか n 階からは下にしかいけないとします。さて、 k 歩いた後に m 階にいる確率は？

この問題はマルコフ連鎖の 1 つで $n=5$ の場合にグラフは以下ようになる。



マルコフ連鎖の確率分布はベクトル $[p_1, p_2, \dots, p_n]$ で p_k は現在の状態が k である確率です。この時 $p_1 + p_2 + \dots + p_n = 1$ は常に成立します。

上記のシナリオでは最初は 1 階にいるので初期分布は $[1, 0, 0, 0, 0]$ です。次の瞬間フロア 1 からフロア 2 にしか移動できないので、次の分布は $[0, 1, 0, 0, 0]$ です。この後は 1 階上か 1 階下に移動できるので次の分布は $[1/2, 0, 1/2, 0, 0]$ となり、以下同様です。

これをシミュレーションする効率的な方法は、動的計画法です。各ステップで、どのように動くことができるかのすべての可能性を調べていけば、 m ステップの歩みを $O(n^2 m)$ 時間でシミュレートすることができます。

また、この遷移は確率分布を更新する行列として表現することもできます。

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

確率分布にこの行列を掛けると、1 ステップ移動した後の新しい分布が得られま

す。例えば、分布 $[1, 0, 0, 0, 0]$ から分布 $[0, 1, 0, 0, 0]$ は次の通りです。

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

行列の累乗を効率的に計算することで m ステップ後の分布を $O(n^3 \log m)$ で計算することができます。

24.5 乱択

問題を解くために確率とは関係ない問題だとしても、ランダム性を利用することがあります。**乱択**はランダム性を用いたアルゴリズムです。

モンテカルロ法 (Monte Carlo algorithm) は間違った答えとなる可能性を十分に持つランダム化アルゴリズムです。このアルゴリズムが適切であるためには、間違った答えが出る確率が十分に小さいことが必要です。

ラスベガス法は、間違った答えを出さないが、実行時間はランダムに変化するアルゴリズムである。このアルゴリズムのデザインには効率的なアルゴリズムの設計が必要です。

ランダム性を利用して解くことができる 3 つの例題を紹介します。

k 番目に小さい数 (Order statistics)

配列の **k 番目に小さい数 (Order statistics)** は昇順にソートした後の位置 k にある要素です。ですが、ある 1 つの要素を見つけるためだけに配列全体をソートする必要があるのでしょうか？ 配列をソートせずにランダムなアルゴリズムでこれを求めることができます。これは **quickselect**^{*1} アルゴリズム、別名ラスベガス・アルゴリズムと呼ばれ、その実行時間は通常 $O(n)$ 、最悪の場合 $O(n^2)$ です。

このアルゴリズムは配列のランダムな要素 x を選び x より小さい要素を配列の左に、それ以外の要素を配列の右側に移動させます。これは要素が n 個とすると $O(n)$ で実行できます。左側には a 個の要素、右側には b 個の要素があるとしましょう。 $a = k$ ならば、要素 x は k 番目に小さい数です。 $a > k$ の時は左側部分の k 番目の数を再帰的に求め、 $a < k$ の時は右側部分の r 番目の数 ($r = k - a$) を再帰的に求め、要素が見つかるまで同様の方法で探索を見つければ良いのです。

^{*1} In 1961, C. A. R. Hoare published two algorithms that are efficient on average: **quicksort** [36] for sorting arrays and **quickselect** [37] for finding order statistics.

x がランダムに選ばれるため、配列のサイズは各ステップで約半分と期待できるので時間計算量は次のようになります。

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

最悪の場合は $O(n^2)$ です。これは x が配列の最小または最大の要素の 1 つになるように常に選択される場合で $O(n)$ ステップが必要になるからです。その確率は非常に小さいので実際にはこのようなことは起こらないでしょう。

行列の乗算の検証 - Verifying matrix multiplication

次の問題は**検証**です。 A, B, C が $n \times n$ である時に、 $AB = C$ が成り立つかどうかを調べます。もちろん、 AB を $O(n^3)$ で計算すれば良いです。これをもっと簡単に求めたいです。

この問題はモンテカルロアルゴリズム^{*2}で解くことができます。この計算量は $O(n^2)$ です。考え方は簡単で n 個の要素の X をランダムなベクトルとして選びます。そして、 ABX と CX を計算する。 $ABX = CX$ なら、 $AB = C$ です。そうでなければ $AB \neq C$ です。

このアルゴリズムの時間計算量は $O(n^2)$ です。 ABX と CX を $O(n^2)$ の時間で計算できるためです。行列 ABX を効率よく計算するには、以下のように工夫できます。 $A(BX)$ という様に計算することで、 $n \times n$ と $n \times 1$ の演算をすれば良いからです。

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

ですが、

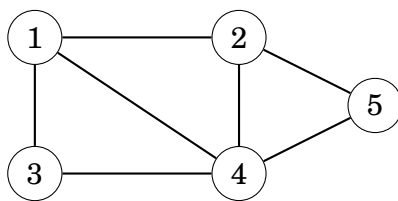
$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

アルゴリズムが誤りを犯す確率は小さいので、 $AB = C$ と報告する前に、さらにいくつかのランダムベクトル X を用いて結果を検証することでこの確率をさらに下げることができます。

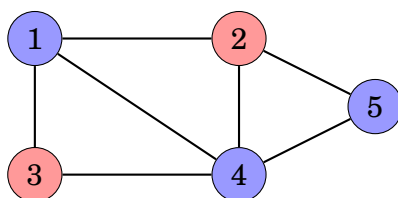
グラフの彩色 - Graph coloring

n 個のノードと m 個のエッジを持つグラフが与えられたとき、少なくとも $m/2$ 個の辺の両端同士が異なる色になるように、グラフのノードを 2 色で彩色する方法を見つける問題です。

^{*2} R. M. Freivalds published this algorithm in 1977 [26], and it is sometimes called **Freivalds' algorithm**.



一例は以下のとおりです。



このグラフには 7 本の辺があり、そのうち 5 本は両端の色が異なるので有効です。

この問題は有効な彩色が見つかるまでランダムに点を選ぶ生成するラスベガス・アルゴリズムを使って解けます。各ノードの色は独立に選ばれ両方の色の確率が $1/2$ になるようにします。

一つの辺の端点異なる色になる確率は $1/2$ なので、端点の色が異なる辺の数の期待値は $m/2$ です。このため何度か試せば成立すると期待できます。

第 25 章

ゲーム理論

この章ではランダムな要素を含まない 2 人用ゲームに焦点を当てます。目的は相手が何をやってもゲームに勝てるような戦略 (もしそのような戦略) を見つけることです。このようなゲームには一般的な戦略があり、Nim を使ってゲームを分析することができます。まずプレイヤーが山から棒を取り除く簡単なゲームを分析し、その後一般化を行います。

25.1 ゲームの状態 - Game states

最初に n 本の棒があるゲームを考えます。プレイヤー A と B は交互に交代します。最初はプレイヤー A からスタートします。各手番で 1 本、2 本、3 本の棒を山から取り除き、最後の棒を取り除いたプレイヤーがゲームに勝つとします。

$n=10$ の場合、次のようにゲームを進めることができます。

- プレイヤー A は 2 本取る (残り 8 本)
- プレイヤー B は 3 本取る (残り 5 本)
- プレイヤー A は 1 本取る (残り 4 本)
- プレイヤー B は 2 本取る (残り 2 本)
- プレイヤー A は 2 本取って勝ち

このゲームは $0, 1, 2, \dots, n$ の状態からなって状態の数は残っているスティックの数に対応します。

勝ち状態と負け状態 - Winning and losing states

勝利状態とは適切に動けば必ず勝ちが確定する状態です。敗北状態とは相手が最適なプレイをすればゲームに負ける状態である。このようにゲームの状態をすべて

分類することで、それぞれの状態が「勝ちの状態」と「負けの状態」のどちらかになることがわかります。

上のゲームを考えます。状態 0 は明らかに負け状態で、プレイヤーは何も手を打てません。状態 1、2、3 は、最後の 1 つを取れるので勝ちの状態です。状態 4 は逆にどの手を打っても相手が勝ち状態になるため負け状態と言えます。一般的には現在の状態から負け状態にできる現在の状態は勝ち状態であり、そうでなければ現在の状態は負け状態と言えます。これを利用して可能な手がない負け状態から始まるゲームのすべての状態を分類することができます。

状態 0...15 を分類します (W は勝ち状態、L は負け状態)。

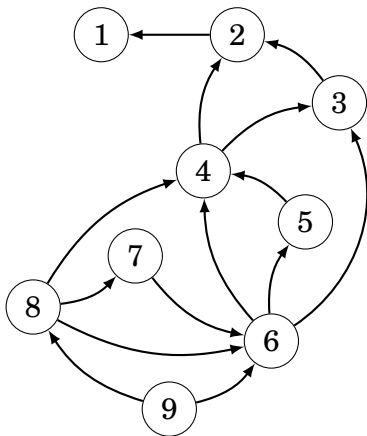
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W

このゲームは簡単で、 k が 4 で割り切れる場合は負け。それ以外は勝ち状態です。このゲームの最適なプレイ方法はスティックが 4 で割り切れる状態にすれば良いです。もちろんこちらが動くときに棒の数が 4 で割り切れないことが条件です。もし 4 で割り切れるなら相手が最適な手を打つ限り勝てません。

状態グラフ - State graph

別のゲームを考えます。各状態 k において x が k より小さく k を割り切れる x 本の棒を取り除けるとします。例えば状態 8 では 1 本、2 本、4 本の棒を取り除くことができ、状態 7 では 1 本の棒を取り除くことだけが許されるとします。

次の図は、状態 1...9 を状態グラフで表したもので、ノードが状態、エッジが取りうる移動です。



このゲームの最終状態は常に状態 1 で、有効な手がないため負け状態です。状態 1...9 の勝ち状態と負け状態は次のとおりです。

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

意外かもしれませんがこのゲームは、偶数の時勝ち状態で奇数の時負け状態です。

25.2 Nim - Nim game

Nim - nim game は単純なゲームですが同じ戦略を用いて他の多くのゲームを行うことができるため、ゲーム理論において重要な考え方です。

Nim には n 個の山があり各山にはある数の棒があります。プレイヤーは交互に移動してまだ棒が入っている山を選んで任意の本数の棒を取り除きます。最後のスティックを取り除いたプレイヤーが勝者です。

Nim の初期状態は $[x_1, x_2, \dots, x_n]$ で与えられ、 x_k は k 個目の山のスティックの数とします。 $[10, 12, 5]$ は 10, 12, 5 のスティックを持つ三つの山がある初期状態です。 $[0, 0, \dots, 0]$ の状態は、棒を 1 本も取り出せないで負け状態であり、これが常に最終状態です。

考察 - Analysis

nim sum s を用いて分析を行うことができます^{*1}。 $s = x_1 \oplus x_2 \oplus \dots \oplus x_n$ として \oplus は XOR 演算です。 s が 0 の時は負けでそれ以外は勝ち状態です。例えば $[10, 12, 5]$ は $10 \oplus 12 \oplus 5 = 3$ なので勝ち状態です。

ですが **nim sum** と **nim** ゲームはどのように関係しているのでしょうか？ **nim** の状態が変化したときに、**nim sum** がどのように変化するかを見ていきます。最終状態 $[0, 0, \dots, 0]$ は負け状態で、その s は 0 です。他の負け状態を考えるとどのような動きも勝ち状態になります。一つの値 x_k が変化すると **nim sum** も変化するので、どんな作業も **nim sum** は 0 と異なる結果に遷移するからです。

勝ち組の状態を考えます。 $x_k \oplus s < x_k$ となる山 k があれば負け状態に遷移できます。つまり勝てます。この場合、山 k からスティックを取り除いて $x_k \oplus s$ のスティックを含むようにすれば負け状態に移行できます。このような山は必ず存在し、いずれかの x_k とは s の左端の 1 ビットのと同じ 1 を持った山です。

$[10, 12, 5]$ の状態を考えてみましょう。この状態は **nim sum** が 3 なので勝ちの状態であり、そのような手を見つけます。

^{*1} The optimal strategy for nim was published in 1901 by C. L. Bouton [10].

10	1010
12	1100
5	0101
<hr/>	
3	0011

この場合、10 本ある山が nim sum の左端の 1 ビットの位置に 1 ビットを持つ唯一の山です。

10	10 <u>1</u> 0
12	1100
5	0101
<hr/>	
3	00 <u>1</u> 1

山の本数は $10 \oplus 3 = 9$ にしたいので 1 本だけ取ります。この後、状態は [9,12,5] となり、負け状態に遷移できます。

9	1001
12	1100
5	0101
<hr/>	
0	0000

misere nim game - Misère game

misere nim game - misère game はゲームの目的が逆です。つまり、最後のスティックを取ったプレイヤーがゲームに負けます。misere nim game は標準の nim とほぼ同じ様に考えられます。

最初は標準的な nim ゲームのようを行うがゲームの終盤に戦略を変えます。戦略を変えるのは、次の手の後に各山が最大で 1 本の棒を含むような状況で導入します。

オリジナルのゲームでは、1 本の棒を持つ山が偶数個になるような手を選ぶべきです。しかし、misère ゲームでは、1 本の棒の奇数個の山ができるようにします。

まず、この戦略が変化する状態が必ずゲームに現れます。この状態になったとき、ちょうど複数のスティックを持つ山が一つ含まれているので、Nim Sum は 0 ではないので勝利が確定します。

25.3 スプレイグ・グランディの定理 - Sprague – Grundy theorem

スプレイグ・グランディの定理 - Sprague – Grundy theorem^{*2}により nim で用いられている戦略は以下の要件を満たすすべてのゲームに一般化されます。

- 2 人のプレイヤーが交互にプレイする
- ゲームは状態から構成される。ある状態において可能な手は誰の手番であるかには依存しない
- プレイヤーが手を出せなくなるとゲーム終了
- ゲームは遅かれ早かれ必ず終わる
- プレイヤーは状態と許容される手について完全な情報を持っておりゲームにランダム性はない

このアイデアは、各ゲーム状態の Grundy 数を nim の山に紐づけて管理することにあります。すべての状態の Grundy 数が分かれば nim ゲームのようにプレイができます。

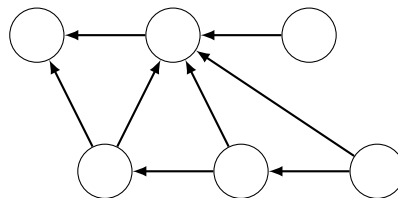
Grundy 数 - Grundy numbers

ゲームの状態の **Grundy 数 - Grundy number** とは次のように示されます。

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

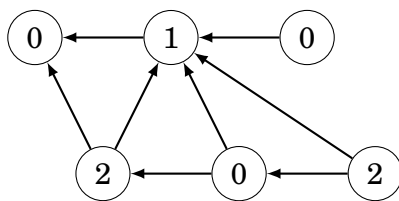
g_1, g_2, \dots, g_n は移動可能な状態の Grundy 数で、 mex 関数はその集合に含まれない最小の非負の数のことです。例えば、 $\text{mex}(\{0, 1, 3\}) = 2$ です。ある状態で移動可能なものがない場合、 $\text{mex}(\emptyset) = 0$ 。なので、そのグランディ数は 0 とします。

例えば、以下を考えます。



この Grundy 数は以下の通りです。

^{*2} The theorem was independently discovered by R. Sprague [61] and P. M. Grundy [31].



負けた状態の Grundy 数は 0 で勝った状態の Grundy 数は正の数です。

ある状態の Grundy 数とは Nim の山のスティックの本数に相当します。Grundy 数が 0 であれば、Grundy 数が正である状態にのみ移動でき、Grundy 数が $x > 0$ であれば、Grundy 数が $0, 1, \dots, x-1$ のすべての数を含む状態に移動することが可能になります。

迷路の中で図形を動かすゲームを考えます。迷路の各マスは床か壁のどちらかです。手番で、プレイヤーは図形を左か上に何歩か移動させなければならないとします。最後に動かしたプレイヤーが勝者です。

次の図はゲームの初期状態の例です。@は図形で*は移動可能なマスを表します。

				*
				*
*	*	*	*	@

ゲームの状態は迷路のすべての床のマス目で表現できます。上の迷路では Grundy 数は以下の通りです。

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

したがって迷路ゲームの各状態は、nim ゲームの山に対応します。右下のマスの Grundy 数は 2 なので、勝利状態です。4 歩左に移動するか 2 歩上に移動するかで負け状態に遷移できるのでゲームに勝つことができます。

このゲームでは本来の nim ゲームとは異なり、現在の状態の Grundy 数よりも大きな Grundy 数を持つ状態に移動も可能です。ただし、相手はそれを打ち消す手を常に選べるので負け状態から脱出することはできません。

サブゲーム - Subgames

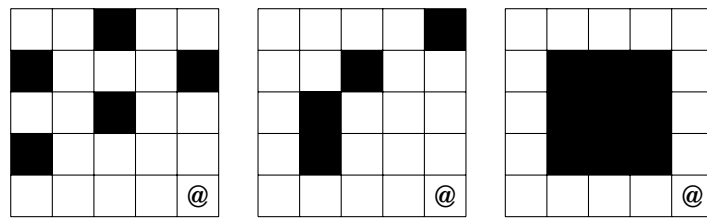
このゲームがサブゲームから構成されていると仮定しましょう。各ターンにおいてプレイヤーはまずサブゲームを選択し、次にサブゲームにおける一手を選択するものとしましょう。ゲームはどのサブゲームでも手を打つことができなくなったときに終了します。

この場合の複数のサブゲームから構成されるあるゲームの Grundy 数はサブゲームの Grundy 数の Nim 和となります。

サブゲームの Grundy 数をすべて計算しその Nim 和を計算することで、Nim ゲームのようにプレイすることができます。

例として3つの迷路のサブゲームで構成されるゲームを考えてみましょう。このゲームでは、プレイヤーは手番ごとに迷路の一つを選び、その迷路の中で図形を移動させられるとします。

ゲームの初期状態を次のように仮定しましょう。



各迷路に置ける Grundy 数は以下の通りになります。

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

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

0	1	2	3	4
1				0
2				1
3				2
4	0	1	2	3

初期状態が $2 \oplus 3 \oplus 3 = 2$ であるため、先攻が勝利します。例えば、最初の1手で2マス上に進めば、 $0 \oplus 3 \oplus 3 = 0$ を生成して相手に渡せます。

Grundy ゲーム - Grundy's game

ある対局の一手が、その対局を互いに独立した下位の対局に分割するとみなせることがあります。この場合のそのゲームの Grundy 数は

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

となり n はゲームの可能な手の数です。

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

k の手順は $a_{k,1}, a_{k,2}, \dots, a_{k,m}$ のサブゲームを生成するとします。

Grundy ゲーム - Grundy's game はこのようなゲームの例です。初期状態で n 本の棒を含む一つの山があります。手番ごとにプレイヤーは山を選び、それを空でない二つの山に分割し、その山の大きさが異なるようにしなければなりません。最後に手を打ったプレイヤーがゲームに勝ちます。

n 本の棒を含むヒープの Grundy 数を $f(n)$ としましょう。Grundy 数は、山を 2 つの山に分割する方法をすべて調べれば計算できます。例えば、 $n=8$ のとき $1+7$, $2+6$, $3+5$ の可能性があるので以下のようになります。

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

このゲームでは、 $f(n)$ の値は、 $f(1), \dots, f(n-1)$ の値に基づいて決定されます。1 本と 2 本の棒の山を分割することはできないので $f(1) = f(2) = 0$ であることに注意します。

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

$n=8$ のときのグランディ数は 2 なのでゲームに勝つことが可能です。 $f(1) \oplus f(7) = 0$ なので勝ち手はヒープ $1+7$ を作ることです。

第 26 章

文字列アルゴリズム - String algorithms

文字列処理の効率的なアルゴリズムを解説します。多くの文字列に関する問い合わせは $O(n^2)$ で容易に解くことができますが、競技プログラミングでは $O(n)$ または $O(n \log n)$ で動作するアルゴリズムが求められることがあります。

例えば長さ n の文字列と長さ m のパターンが与えられたとき、文字列中にそのパターンが何回現れるかを探すというパターンマッチの問題があります。ABC というパターンは、ABABCBABC という文字列の中に 2 回出現します。

パターンマッチング問題は、文字列中のパターンが出現する可能性のあるすべての位置をテストするブルートフォースアルゴリズムにより $O(nm)$ で容易に解けます。これを $O(n+m)$ で処理できる効率的なアルゴリズムがあることを見ていきましょう。

26.1 文字列に関する用語 - String terminology

この章では文字列には 0-indexies が使用されるとします。つまり、長さ n の文字列 s は、文字 $s[0], s[1], \dots, s[n-1]$ のことです。文字列の中に現れる可能性のある文字の集合をアルファベットとします。 $\{A, B, \dots, Z\}$ がアルファベットの例です。

部分文字列 - substring は文字列の中の連続した文字の並びのことです。 $s[a \dots b]$ という変数を考えた時に文字列 s の a から開始して b 文字を含む部分の文字列を考えます。長さ n の文字列は $n(n+1)/2$ の部分文字列を持ちます。例えば、ABCD の部分文字列は A, B, C, D, AB, BC, CD, ABC, BCD, ABCD になります。

部分配列 - subsequence は、連続でなくてもよい文字を元の順序を保持しながら並べたものです。長さ n の文字列は $2^n - 1$ の部分列を持ちます。ABCD の部分配

列は, A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD, ABCD となります。

プレフィックス - prefix は文字列の最初からの任意の長さの部分文字列です。**サフィックス - suffix** は文字列の最後の文字を含むの任意の長さの部分文字列です。ABCD のプレフィックスは A, AB, ABC, ABCD です。ABCD のサフィックスは D, CD, BCD, ABCD です。

回転 - rotation は、文字列の文字を 1 つずつ先頭から最後へ (またはその逆に) 移動させることで生成することができる文字列です。ABCD の回転は ABCD, BCDA, CDAB, DABC です。

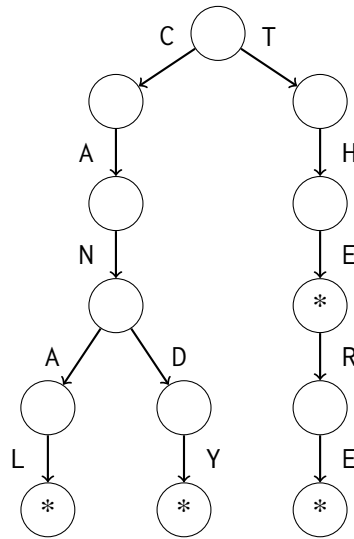
ピリオド - period とは文字列の接頭辞であって、そのピリオドを繰り返すことで文字列を構成することができるものです。最後の繰り返しは部分的で、ピリオドの接頭語だけを含むこともあります。ABCABCA の最も短いピリオドは ABC です。(訳註: 最後の A のみの文字列はピリオドの 1 文字目で構成されています)

ボーダー - border は、ある文字列の接頭辞と接尾辞の両方を持つ文字列のことです。ABACABA のボーダーは A, ABA, ABACABA となります。

文字列における**辞書順 - lexicographical order** はアルファベットの順序で比較されます。その 2 つは異なることを前提として ($x \neq y$)、文字列が $x < y$ である条件は次の 2 つのどちらかです。一つは x が y のプレフィックスである時。あるいは、 $x[k] < y[k]$ であるような k が存在し $i < k$ であるような i に対して $x[i] = y[i]$ である時、です。

26.2 トライ構造 - Trie structure

trie とは文字列の集合を保持するような根付きの木です。集合内の各文字列を根を起点とする文字のパスとして格納します。2 つの文字列が共通の接頭辞を持つ場合、それらはツリー内で共通のパスを持ちます。例えば次のようなトライの例を示します。



このトライは、{CANAL,CANDY,THE,THERE} を対応しています。ノードにある文字*は、集合の中の文字列がそのノードで終わることを示します。このような情報がが必要なのは、ある文字列が他の文字列の接頭辞である場合があるためです。たとえば、上のトライでは THE は THERE のプレフィックスです。

根から始まるパスをたどればよいのでトライが長さ n の文字列を含むかどうかは $O(n)$ 時間で確認できます。また、長さ n の文字列をトライに追加するにはパスを辿っていき必要ならトライに新しいノードを追加すればよいので $O(n)$ で可能です。

トライを用いると与えられた文字列のうち、その接頭辞が集合に属する最長のプレフィックスを求めることができます。さらに各ノードに追加情報を格納することで、集合に属してかつ、与えられた文字列を接頭辞として持つ文字列の数を計算することができます。

トライは配列に格納することができます。

```
int trie[N][A];
```

ここで、 N はノードの最大数 (文字列の最大合計長)、 A はアルファベットの大きさです。トライのノードはルートの番号が 0 になるように $0,1,2,\dots$ と番号付けを行い、 $trie[s][c]$ はノード s から文字 c を使って移動したときの次のノードを示します。

26.3 文字列のハッシュ化 - String hashing

文字列のハッシュ化 - **String hashing** は 2 つの文字列が等しいかどうかを効率的にチェックするためのテクニック^{*1} です。文字列ハッシュは文字単位ではなく文字列全体のハッシュ値を用います。

ハッシュ値の計算 - Calculating hash values

文字列のハッシュ値 - **hash value** は文字列全体の文字から算出される数値のことです。決まったルールで算出を行うので 2 つの文字列が同じであればハッシュ値も同じになり、ハッシュ値をもとに文字列を比較することができます。通常、この実装方法は多項式ハッシュであり長さ n の文字列 s のハッシュ値として

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B,$$

と示され、ここで、 $s[0], s[1], \dots, s[n-1]$ は s の各文字を示し、 A と B はあらかじめ選択した定数となります。

例えば文字列 ALLEY を考えます。

A	L	L	E	Y
65	76	76	69	89

$A = 3$ 、 $B = 97$ とするとこのハッシュ値は

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

前処理 - Preprocessing

多項式ハッシュを用いることで文字列 s の任意の部分文字列のハッシュ値を $O(n)$ で求めて、この前処理を行った後は $O(1)$ 時間で計算することができます。このアイデアは $h[k]$ が接頭辞 $s[0 \dots k]$ のハッシュ値を含むような配列 h を構築していることです。配列の値は以下のように再帰的に計算することができます。

$$\begin{aligned} h[0] &= s[0] \\ h[k] &= (h[k-1]A + s[k]) \bmod B \end{aligned}$$

さらに $p[k] = A^k \bmod B$ となる配列 p を構築します。

$$\begin{aligned} p[0] &= 1 \\ p[k] &= (p[k-1]A) \bmod B. \end{aligned}$$

^{*1} The technique was popularized by the Karp – Rabin pattern matching algorithm [42].

これらの配列の構築は $O(n)$ で行えます。この後に任意の部分文字列 $s[a \dots b]$ のハッシュ値は $a > 0$ である場合、以下の式を用いて $O(1)$ で計算することができます。

$$(h[b] - h[a - 1]p[b - a + 1]) \bmod B$$

$a = 0$ の場合は単に $h[b]$ となります。

ハッシュ値の利用 - Using hash values

このような文字列のハッシュ値を使うと文字列を効率的に比較することができます。ハッシュ値が等しければ、文字列はおそらく等しく (訳注: ハッシュの衝突の可能性がゼロとはいえないのでおそらくとなっています)、ハッシュ値が異なれば、文字列は確実に異なります。

ハッシュ化の活用としてブルートフォースアルゴリズムを効率化できるケースがあります。例えば文字列 s とパターン p が与えられたとき s の中で p が出現する位置を見つけるというパターンマッチング問題を考えます。ブルートフォースアルゴリズムは p が出現しそうな位置をすべて調べ、文字列を 1 文字ずつ比較する手法ですが、このようなアルゴリズムの時間計算量は $O(n^2)$ です。

これをハッシュ値を使用することでより効率的にすることができます。ハッシュを用いると部分文字列のハッシュ値のみが比較されるため各比較は $O(1)$ しかかりません。この結果、時間計算量 $O(n)$ のアルゴリズムとなってとても良い時間計算量となります。

ハッシュと二分探索を組み合わせることで、2 つの文字列の辞書的順序を対数時間で求めることも可能です。2 つの文字列の共通接頭辞の長さを二分探索で計算します。こうして共通の接頭辞の長さがわかればあとは接頭辞の次の文字を ($O(1)$ で) 調べればよいのです。

衝突と定数パラメータ - Collisions and parameters

ハッシュ値を利用する際にはハッシュ値の衝突 - **collision** を考慮しないとなりません。ハッシュ値に依存するアルゴリズムは、もしハッシュ値の衝突が発生してしまうと、2 つの文字列が等しいと判定してしまいます。

一般的に存在する文字列の数はハッシュ値で表現できる数より大きいので、衝突は常に起こりうる問題です。しかし、定数 A 、 B を上手く選べば衝突の確率は小さくできます。通常の方法は、 10^9 に近いランダムな定数を以下のように選ぶことです。

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

この定数を使うと、ハッシュ値を計算するときに、 AB と BB の積が long long に収まるので、long long 型が使えます。さて、ハッシュ値は 10^9 くらいあれば十分なのでしょうか？

次の3つのシナリオを考えてみましょう。

Scenario 1: 文字列 x と y を比較する。衝突の確率は、全てのハッシュ値が等確率であると仮定すると $1/B$ となる。

Scenario 2: 文字列 x を複数の文字列 y_1, y_2, \dots, y_n と比較する。この時の1つ以上の衝突の確率は次の通り。

$$1 - \left(1 - \frac{1}{B}\right)^n.$$

Scenario 3: 複数の文字列 x_1, x_2, \dots, x_n を全て互いに比較したとする。この時の1つ以上の衝突の確率は次の通り。

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

$n = 10^6$ として b を変化させた衝突確率を以下に示します。

定数 B	scenario 1	scenario 2	scenario 3
10^3	0.001000	1.000000	1.000000
10^6	0.000001	0.632121	1.000000
10^9	0.000000	0.001000	1.000000
10^{12}	0.000000	0.000000	0.393469
10^{15}	0.000000	0.000000	0.000500
10^{18}	0.000000	0.000000	0.000001

表から、シナリオ 1 では、 $B \approx 10^9$ のとき、衝突の確率は無視できる程度です。シナリオ 2 では、衝突の可能性はあるが、その確率はまだかなり小さいです。しかし、シナリオ 3 では状況は大きく異なり、 $B \approx 10^9$ のとき、かなりの確率で衝突が起こります。

シナリオ 3 の現象は、**誕生日のパラドックス -birthday paradox** として知られている問題です。 n 人いれば、 n がかなり小さくても同じ誕生日の人がいる確率は大きくなります。これがハッシュでも同じことが言えます。

そこで、異なるパラメータを用いて**複数**のハッシュ値を計算することで、衝突の確率を小さくすることができます。この場合にすべてのハッシュ値で同時に衝突が発生することは考えにくくなります。例えばパラメータ $B \approx 10^9$ のハッシュ値 2 個は、パラメータ $B \approx 10^{18}$ のハッシュ値 1 個に相当するため衝突の可能性を格段に下げることができます。

定数として $B = 2^{32}$ と $B = 2^{64}$ を選択する人もいます。これは、 2^{32} や 2^{64} での mod をとると 32bit 長や 64bit 長の型で計算できるように思えます。しかし、これは良い選択ではありません。 2^x の形の定数は衝突を発生させる入力を作成することが可能だからです [51]。

26.4 Z-アルゴリズム - Z-algorithm

長さ n の文字列 s の **Z 配列 - Z-array** である z は、 $k = 0, 1, \dots, n-1$ に対して、 k の位置から始まる s の最長の部分文字列の長さと s の接頭辞を含みます。したがって、 $z[k] = p$ は $s[0 \dots p-1]$ が $s[k \dots k+p-1]$ に等しいことを意味します。文字列処理の問題の多くは、Z 配列を使うと効果的に解くことができます。

ACBACDACBACBACDA の Z 配列の例を示します。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
—	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

このケースでは $z[6] = 5$ となります。部分文字列 ACBAC は長さ 5 の s のプレフィックスですが、部分文字列 ACBACB という長さ 6 の s のプレフィックスではないからです。

Z アルゴリズムの説明 - Algorithm description

これを求めるアルゴリズムを紹介していきます。**Z アルゴリズム - Z-algorithm**^{*2} と呼ばれる、Z 配列を $O(n)$ 時間で効率的に構成するアルゴリズムがあります。このアルゴリズムは、Z 配列に既に格納されている情報を利用し、また、文字列を一文字ずつ比較することで Z 配列の値を左から右へと計算していきます。

Z 配列の値を効率的に計算するために、アルゴリズムは $s[x \dots y]$ が s の接頭辞で、 y ができるだけ大きくなるような区間 $[x, y]$ を維持します。 $s[0 \dots y-x]$ と $s[x \dots y]$ は等しいことが分かっているので、位置 $x+1, x+2, \dots, y$ の Z 値を計算するときはこの情報を利用できます。

各位置 k で $z[k-x]$ の値を確認します。 $k+z[k-x] < y$ ならば、 $z[k] = z[k-x]$ です。しかし、 $k+z[k-x] \geq y$ の場合は $s[0 \dots y-k]$ は $s[k \dots y]$ と等しく、 $z[k]$ の値を決定するためには、部分文字列の文字列を比較する必要があります。このアルゴリズムは $O(n)$ で動作します。なぜなら、比較が $y-k+1$ と $y+1$ で行われるため

^{*2} The Z-algorithm was presented in [32] as the simplest known method for linear-time pattern matching, and the original idea was attributed to [50].

です。
例えば、次のような Z 配列を考えます。


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

$z[6] = 5$ を計算し $[x,y]$ のレンジは $[6,10]$ となります。

						x			y						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?


これで $s[0...4]$ と $s[6...10]$ が等しいことがわかったので、以降の Z 配列の値を効率的に計算できるようになります。まず、 $z[1] = z[2] = 0$ なので $z[7] = z[8] = 0$ もすぐにわかります。

						x			y						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



ここで $z[3] = 2$ から $z[9] \geq 2$ がわかります。

						x			y						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



10 以降の文字列については情報がないので文字ごとに部分文字列を比較します。

						x			y						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

$z[9] = 7$ となったので新しい $[x, y]$ は $[9, 15]$ となります。

									$x \qquad \qquad \qquad y$						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	0	0	7	?	?	?	?	?	?

あとは、Z 配列の値を使って他の Z 配列の文字列を決めることができます。

									$x \qquad \qquad \qquad y$						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

Z 配列の利用シーン - Using the Z-array

文字列ハッシュと Z-アルゴリズムのどちらを使うかは、好みの問題であるケースが多いです。ハッシュとは異なり、Z-アルゴリズムには衝突の危険性はありません。一方、Z-アルゴリズムは実装が難しく、また、いくつかの問題はハッシュを使うことでしか解決できません。

試しに文字列 s の中にあるパターン p の出現を見つけるというパターンマッチングの問題を考えます。すでに文字列ハッシュを使って解決しましたが、Z-アルゴリズムでの解決方法があります。

Z アルゴリズムで文字列処理するためには、特殊文字で区切られた複数の文字列からなる 1 つの文字列を構成するのが一般的な考え方です。この問題では、 p と s の文字列中に出現しない特殊文字 $\#$ で区切った文字列 $p\#s$ を構築します。 $p\#s$ の Z 配列は、 s の中で p が出現する位置を示すことになり、その位置は p の長さを含んでいるためです。

例えば、 $s = \text{HATTIVATTI}$ で $p = \text{ATT}$ としましょう。

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	T	T	#	H	A	T	T	I	V	A	T	T	I
-	0	0	0	0	3	0	0	0	0	3	0	0	0

5,10 には長さ 3 が含まれているので、ATT が HATTIVATTI のこの位置で発生していることがわかります。

Z 配列の計算には線形時間がかかるため、この計算は線形で実行することができます。

実装 - Implementation

ここでは、Z 配列を返す Z アルゴリズムの実装を示します。

```
vector<int> z(string s) {  
    int n = s.size();  
    vector<int> z(n);  
    int x = 0, y = 0;  
    for (int i = 1; i < n; i++) {  
        z[i] = max(0, min(z[i-x], y-i+1));  
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {  
            x = i; y = i+z[i]; z[i]++;  
        }  
    }  
    return z;  
}
```

第 27 章

平方根アルゴリズム - Square root algorithms

平方根アルゴリズム (square root algorithm) は時間計算量が平方根になるアルゴリズムの総称です。平方根は「貧乏人の対数 (poor man's logarithm)」です。 $O(\sqrt{n})$ は $O(n)$ より高速に動作しますが $O(\log n)$ より遅いです。とはいえ多くの平方根アルゴリズムは高速に動作し、競技プログラミングでは実際に使用することができます。

例を挙げて考えます。配列に対してある index の要素を変更する操作と、与えられた区間の合計を計算する操作の 2 つをサポートするデータ構造を考えます。この問題は本書でもみてきた通り、バイナリインデックス木やセグメントツリーを用いることで $O(\log n)$ で両方の操作で解くことができます。ここでは $O(1)$ 時間で要素を修正し $O(\sqrt{n})$ で合計を計算できる平方根アルゴリズムを使ってこの問題にアプローチします。

これは、配列をサイズ \sqrt{n} のブロックに分割し、各ブロックにブロック内の要素の合計が含まれるようにします。例えば、 $n = 16$ 要素の配列は、以下のように 4 要素のブロックに分割されます。

21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

配列要素の変更は、変更のたびに 1 ブロックの合計を更新すればよいので $O(1)$ で行えます。次の図のように値自身とブロックの値を考えれば良いです。

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

さて、ある区間の要素の総和を計算するために、区間を 3 つに分割します。両端にはみ出した要素が両端、と、その間のブロックの 3 つで、この総和を求めることにします。

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

単体の要素数は $O(\sqrt{n})$ 、ブロック数も $O(\sqrt{n})$ であるからであるため、和の問い合わせは $O(\sqrt{n})$ となります。ブロックサイズ $O(\sqrt{n})$ とした目的は配列が $O(\sqrt{n})$ 個のブロックに分割され、各ブロックも $O(\sqrt{n})$ 個の要素を含むという 2 つを達成することです。

\sqrt{n} の値は正確である必要はありません。例えば、パラメータ k を用いて k が \sqrt{n} と異なる n/k を用いてもよいです。最適なパラメータは問題や入力に依存します。例えばブロック全体は頻繁に参照されるがブロックの中の要素を見ることがほとんどないなら、配列を $k < \sqrt{n}$ ブロックに分割してそれぞれが $n/k > \sqrt{n}$ の要素を含むようにするとよいかもしれません。

27.1 アルゴリズムの組み合わせ - Combining algorithms

ここでは、2 つのアルゴリズムを組み合わせた平方根アルゴリズムについて説明します。それぞれのアルゴリズムは片方だけでも $O(n^2)$ 時間で問題を解くことができます。しかしアルゴリズムを組み合わせることで時間計算量は $O(n\sqrt{n})$ となります。

ケース処理 - Case processing

n 個のセルを含む 2 次元の表が与えられたとする。各セルには文字が割り当てられており、距離が最小となる同じ文字を持つ 2 つのセルを見つけないとします。この問題での距離は (x_1, y_1) と (x_2, y_2) の 2 点に対して $|x_1 - x_2| + |y_1 - y_2|$ と定義します。

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

この場合、2 つの'E' 文字の間の距離は 2 です。

各文字を別々に考えることで問題を解くことができます。こう考えると固定文字 c を持つ 2 つのセル間の最小距離を計算することがもんだいです。このための 2 つのアルゴリズムを考えます。

アルゴリズム 1: 文字 c を持つ全てのセルのペアを調べておき、そのセル間の最小距離を計算する。これには $O(k^2)$ の時間がかかる。ここで、 k は文字 c を持つセルの数。

アルゴリズム 2: 文字 c を持つ各セルから同時に開始する幅優先探索。文字 c を持つ 2 つのセル間の最小距離は $O(n)$ で実行できる。

どちらかのアルゴリズムだけをすべての文字に対してそれを使用することでこの問題は解けます。

まず、アルゴリズム 1 を使用する場合、すべてのセルに同じ文字が含まれる可能性があるため、実行時間は $O(n^2)$ となります。 $(k = n$ となるためです)

アルゴリズム 2 を使用する場合、すべてのセルに異なる n 文字が含まれる可能性があるためそれぞれに対して $O(n)$ でクエリするため実行時間は $O(n^2)$ となります。

しかし、2 つのアルゴリズムを組み合わせることで各文字がグリッドに何回現れるかを事前に計算し、文字ごとにアルゴリズムを使い分けることができます。ある文字 c が k 回出現すると仮定しましょう。 $k \leq \sqrt{n}$ ならアルゴリズム 1 を、 $k > \sqrt{n}$ ならアルゴリズム 2 を使います。こうすることで、アルゴリズムの総実行時間は $O(n\sqrt{n})$ となることがわかります。これをもう少し見ていきます。

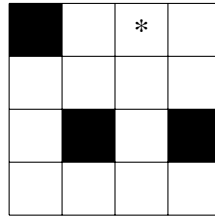
まず、文字 c に対してアルゴリズム 1 を使うとします。 \sqrt{n} 回以下出現する文字 c のセルと他のセルを比較します。この回数は最大で $O(n\sqrt{n})$ 回です。したがって、このようなすべてのセルの処理に使われる時間は $O(n\sqrt{n})$ です。

次に、文字 c に対してアルゴリズム 2 を使用することを考えます。このような文字は最大で \sqrt{n} 個なので、それらの文字の処理には $O(n\sqrt{n})$ 個の時間がかかります。

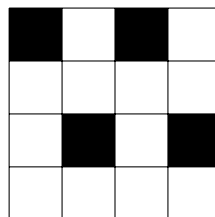
Batch processing

次の問題も n 個のセルを含む 2 次元のグリッドを扱います。初期状態ではまず 1 つを除くセルは白です。これに $n-1$ 回の操作を行います。操作は、まず与えられた白いセルから黒いセルまでの最小距離を計算し、次にその白いセルを黒にします。

例を示します。



この場面で * のついた白いセルから黒いセルまでの最小距離を計算したいとします。2 つ左に移動すれば黒に移動できるので最小距離は 2 です。そして、白いセルを黒く塗ります。



ここでも 2 つのアルゴリズムを考えましょう。

アルゴリズム 1: 幅優先探索を用いて、各白セルについて、最も近い黒セルまでの距離を計算する。これには $O(n)$ の時間がかかり、探索後は任意の白セルから黒セルまでの最小距離を $O(1)$ の時間で求められる。

アルゴリズム 2: 黒く塗られたセルのリストを保持しておき、操作のたびにこのリストをクエリして、また、リストに新しいセルを追加していく。操作には $O(k)$ かかります (k はリストの長さ)。

上記のアルゴリズムを組み合わせて、演算を $O(\sqrt{n})$ のバッチに分割します。各バッチは $O(\sqrt{n})$ 個の操作で構成されます。各バッチの始めに、アルゴリズム 1 を実行します。その後、アルゴリズム 2 を用いて、バッチ内の演算を処理します。バッチとバッチの間にアルゴリズム 2 のリストをクリアします。各バッチでの演算で、黒セルまでの最小距離は、アルゴリズム 1 で計算された距離かアルゴリズム 2 で計算された距離のどちらかになります。

その結果アルゴリズムは $O(n\sqrt{n})$ 時間で動作します。まず、アルゴリズム 1 は $O(\sqrt{n})$ 回実行され各検索は $O(n)$ 時間です。次に、バッチでアルゴリズム 2 を使う場合、リストには $O(\sqrt{n})$ 個のセルが含まれ (バッチの間に リストをクリアするため)、各操作には $O(\sqrt{n})$ 個の時間かかります。

27.2 整数のパーティション - Integer partitions

平方根アルゴリズムは次の考察に基づき使われることも多くあります。正の整数 n を正整数の和で表現した場合、その和は常に最大 $O(\sqrt{n})$ 個の異なる数を含む。

という事実です。さて、最大数の異なる数を含む和を構成するためには、小さな数から選んでいきます。今 $1, 2, \dots, k$ という数を選ぶと得られる総和は

$$\frac{k(k+1)}{2}.$$

です。従って数の最大量は $k = O(\sqrt{n})$ となります。次にこの観測を利用して効率的に解くことができる 2 つの問題を見ていきましょう。

ナップザック問題 - Knapsack

重さの和が n である整数の重さのリストが与えられたとしましょう。我々のタスクは重みの部分集合を使って作れるすべての重さの数を見つけることです。例えば入力が $1, 3, 3$ の場合、可能な和は以下の通りになります。

- 0 (何も選ばない)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

標準的なナップザック問題として捉えると (7.4 章参照)、この問題は次のように解けます。最初の k 個の重みを用いて和 x を形成できる場合に 1、それ以外は 0 である関数 $\text{possible}(x, k)$ を考えます。重さの和が n であることから、重みは最大 n であるため、関数のすべての値は動的計画法を用いて $O(n^2)$ で計算することができます。

しかし、最大で $O(\sqrt{n})$ 個の異なる重りが存在することを利用してアルゴリズムをより効率的にすることができます。まず、重りを同じお守りで構成されるグループに分けて処理します。各グループを $O(n)$ 時間で処理することができて $O(n\sqrt{n})$ とすることができます。

このアイデアは、これまでに処理したグループを使って形成できる重みの和を記録した配列を使用します。配列は n 個の要素からなり、和 k が形成できる場合に 1、そうでない場合に 0 となる配列です。重りのグループを処理するために、配列を左から右へ走査し、このグループと前のグループを使って形成できる新しい重みの和を記録します。

文字列構築 - String construction

長さ n の文字列 s と全ての長さをたしあわせた長さ m の文字列の集合 D があるとき、 s が D 中の文字列の連結して何通りの組み合わせ方があるかを数える問題を考えましょう。例えば $s = \text{ABAB}$ で $D = \{A, B, AB\}$ とすると、次の 4 通りが考えられます。

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

動的計画法を用いてこの問題を解くことができます。ここで $\text{count}(k)$ を D の文字列を用いて接頭辞 $s[0 \dots k]$ を構成する方法の数とすると、 $\text{count}(n - 1)$ が問題の答えとなります。これは、Trie 木を用いて $O(n^2)$ でこの問題を解くことができます。

しかし、文字列のハッシュ値と D に含まれる文字列の長さが最大 $O(\sqrt{m})$ 個であることを利用すれば、この問題をより効率的に解決することができます。

まず、 D に含まれる文字列の全てのハッシュ値を含む集合 H を構築しましょう。

次に、 $\text{count}(k)$ の値を計算する際に、 D に長さ p の文字列が存在するような p の値を全て調べておき、 $s[k - p + 1 \dots k]$ のハッシュ値を計算して、それが H に属するかどうか確認します。

文字列の長さは最大でも $O(\sqrt{m})$ 個なので、実行時間が $O(n\sqrt{m})$ のアルゴリズムとなりました。

27.3 Mo のアルゴリズム - Mo's algorithm

Mo's algorithm^{*1}は、静的な配列の区間クエリに応答し、つまり、配列の値が変化しない問題で 사용할 ことができます。各クエリでは、区間 $[a, b]$ が与えられ、 a と b の間の配列要素に基づく値を計算する必要があるとしましょう。配列は静的なので、クエリは任意の順序で処理できます。Mo のアルゴリズムは、アルゴリズムが効率的に動作することが保証される特別な順序で、クエリを処理していきます。

このアルゴリズムは配列の有効区間 (active range) を保持しておき、有効区間の問い合わせの答えはすぐににわかるようになっています。このアルゴリズムはその有効区間に基づいて問い合わせを処理し、要素を挿入・削除することで有効区間の

^{*1} According to [12]

端点を移動させます。このアルゴリズムの時間計算量は $O(n\sqrt{n}f(n))$ です。ここで、配列は n 個の要素を含み、 n 個の問い合わせがあり、要素の挿入と削除にはそれぞれ $O(f(n))$ の時間がかかるとしましょう。

Mo のアルゴリズムのトリックは、クエリの処理順序です。配列は $k = O(\sqrt{n})$ 要素のブロックに分割されクエリ $[l_1, r_1]$ は、以下のいずれかの場合にクエリ $[l_2, r_2]$ の前に処理されます。

- $\lfloor l_1/k \rfloor < \lfloor l_2/k \rfloor$ または
- $\lfloor l_1/k \rfloor = \lfloor l_2/k \rfloor$ かつ $r_1 < r_2$.

このように、左端があるブロックにある全てのクエリは、右端の順にソートされて次々と処理されます。この順序を用いると、TODO ここ文章直す左端は $O(\sqrt{n})$ ステップで $O(n)$ 移動し、右端は $O(n)$ ステップで $O(\sqrt{n})$ 移動し、アルゴリズム全体は $O(n\sqrt{n})$ の演算しか実行しないことになります。このように、両端点はアルゴリズム中に合計 $O(n\sqrt{n})$ ステップ移動することになります。

例

配列の区間に対応するクエリの集合が与えられて、区間内の異なる要素の数を数えるクエリに答える問題を考えて見ます。Mo アルゴリズムでは、クエリーは常に同じ方法でソートされますが、クエリの答えがどのようにストアされるのかは問題に依存します。この問題では、 $\text{count}[x]$ は要素 x がアクティブ区間に出現する回数を示す配列 count を保持します。あるクエリから別のクエリに移動すると、アクティブな区間を変更されます。例えば、現在の区間が

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

で、次の区間が次の通りだったとします。

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

左の端点が右に 1 ステップ、右の端点が右に 2 ステップ移動することになりますね。各ステップの後、配列 count を更新する必要があります。この更新で $\text{count}[x] = 1$ になった場合、クエリに対する答えは 1 増やし、 $\text{count}[x] = 0$ になった場合、クエリに対する答えも 1 つ減らします。この問題では、各ステップの実行に必要な時間は $O(1)$ であるから、アルゴリズムの総時間複雑度は $O(n\sqrt{n})$ です。

第 28 章

発展的なセグメントツリー - Segment trees revisited

セグメントツリーは多くのアルゴリズム問題の解決に利用できる汎用性の高いデータ構造です。ただしセグメントツリーには、まだ触れていないトピックがたくさんあります。ここでは、より高度なセグメントツリーについて説明します。9.3 章でのセグメントツリーの演算は木の下から上へ歩くように実装してきました。例えば以下のように区間和を計算しました。

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

しかし、より高度なセグメントツリーでは別の方法で上から下へ操作を実装するケースがあります。こうした場合は以下のように書けます。

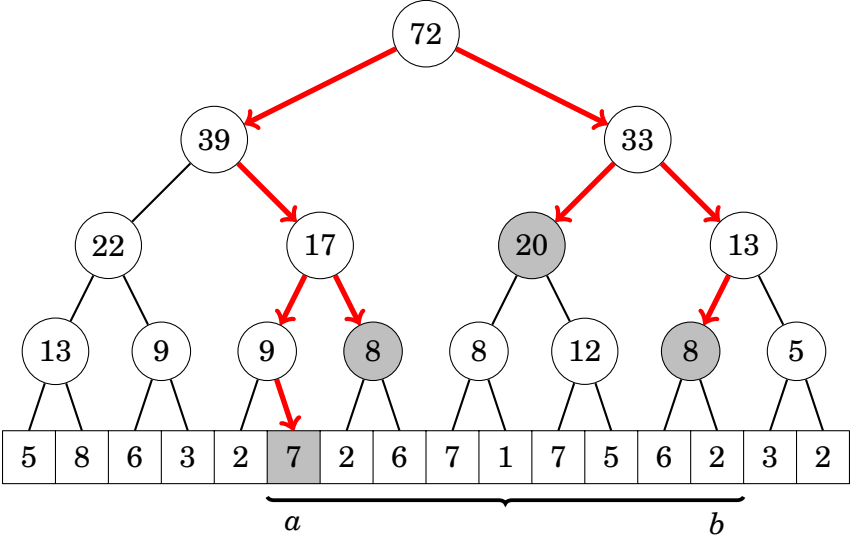
```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

これで、 $[a, b]$ の和を求める $\text{sum}_q(a, b)$ は次のように書けます。

```
int s = sum(a, b, 1, 0, n-1);
```

パラメータ k は、木における現在の位置です。最初は木の根から始まるので、 k は 1 に等しいです。 $[x, y]$ の区間は k に対応し、初期値は $[0, n-1]$ です。和を計算するとき、 $[x, y]$ が $[a, b]$ の外にあれば和は 0 であり、 $[x, y]$ が完全に $[a, b]$ の中にあれば和は自分の木の配下に包含されています。もし $[x, y]$ が $[a, b]$ の部分的な内側にいる時は $[x, y]$ の左右半分への再帰的に探索を行います。左半分を $[x, d]$ 、右半分の $[d+1, y]$ として $d = \lfloor \frac{x+y}{2} \rfloor$ とします。

$\text{sum}_q(a, b)$ を求めるときの検索の進め方を下図に示します。灰色のノードは再帰が停止し、tree で sum を見つけることができるノードを示しています。



この実装では訪問ノード数が $O(\log n)$ であるため計算量は $O(\log n)$ です。

28.1 遅延伝搬 - Lazy propagation

遅延伝搬 - **lazy propagation** を使用すると区間のクエリと更新の両方を $O(\log n)$ でサポートするセグメントツリーが出来ます。このアイデアは更新とクエリを上から下に向かって実行して必要なときだけツリーの下に伝搬されるように更新を遅延させて実行することです。

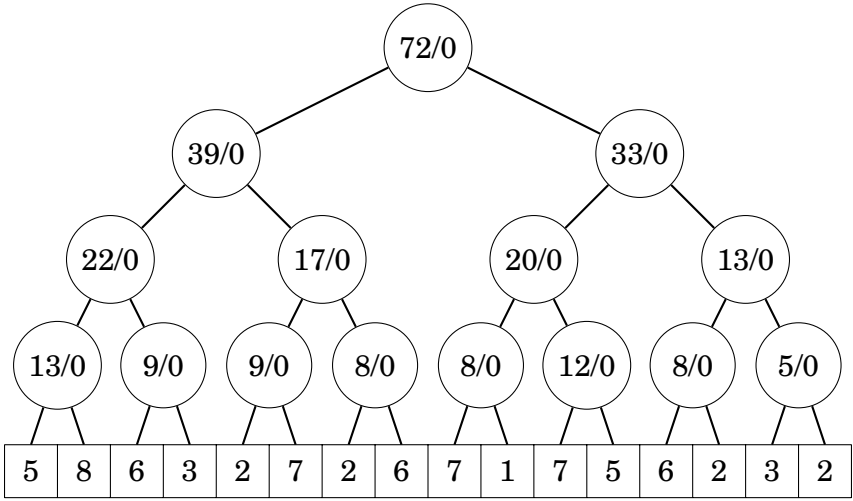
遅延セグメントツリーではノードは 2 種類の情報を含みます。まず通常のセグメントツリーと同様に各ノードには対応する部分木に関連する合計値やその他の値が格納されます。さらに、そのノードの子には伝搬されていない遅延更新に関連する情報を持ちます。

区間更新には 2 種類あり全てのノードの値を**増加**させたいか全てのノードの値に**更新**を行いたいからです。どちらの操作も似たような考え方で実装でき、両方の操作を同時にサポートするツリーを構成することも可能です。

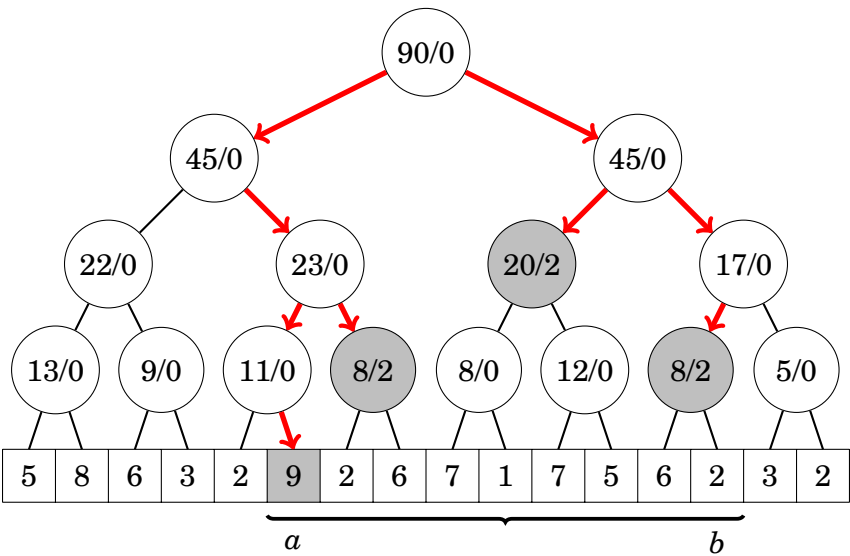
遅延セグメントツリー - Lazy segment trees

例えば、 $[a, b]$ の区間更新 (増加) と区間クエリ (和) の 2 つの操作をサポートするセグメントツリーを構築することを目標とする場合を考えていきます。

各ノードが 2 つの値 s/z を持つ木を構成します。 s は区間内の値の合計を表して z は区間内のすべての値を z だけ増やすべきという遅延更新の値です。以下の木は全ノードで $z = 0$ なので進行中の遅延更新は存在していない状態です。



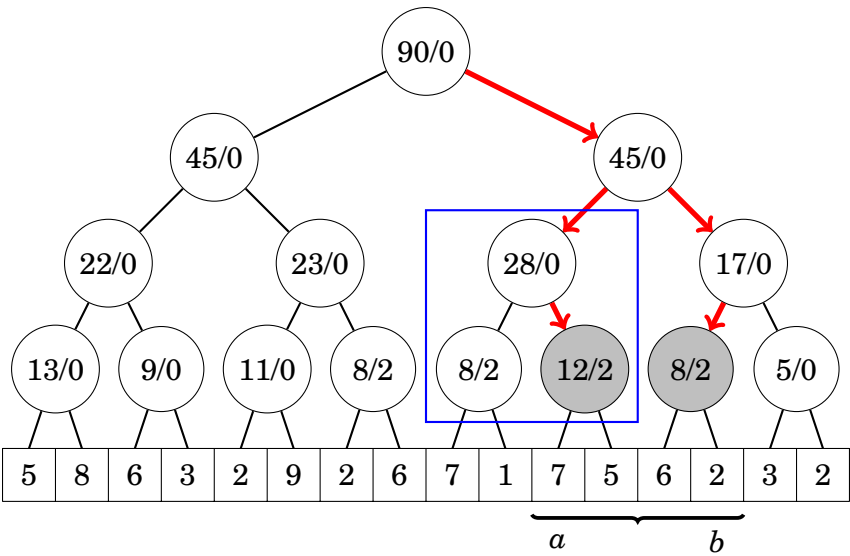
$[a, b]$ の要素が u だけ増やしたいときには根から葉に向かって次のように木のノードを修正していきます。あるノードの区間 $[x, y]$ が完全に $[a, b]$ の内部にある場合、そのノードの z 値を u だけ増加させ停止しましょう。 $[x, y]$ が部分的にしか $[a, b]$ に属さない場合にはノードの s 値を hu (h は $[a, b]$ と $[x, y]$ の交点のサイズ) だけ増やし、木の中で再帰的に辿ることを続けます。下図は $[a, b]$ の要素を 2 増やした後のツリーです。



この遅延の状態を持った木へのクエリ (和) を考えます。区間 $[a,b]$ の区間を木の上から下へ歩いて計算します。あるノードの区間 $[x,y]$ が完全に $[a,b]$ に属する場合、そのノードの s 値を総和に加えます。そうでなければ、木の下方向に向かって再帰的に探索を続ける。ということを繰り返します。

遅延を用いる更新とクエリの両方で、遅延更新の値は常にノードを処理する前にその子ノードに伝搬させます。これは遅延セグメントツリーの重要な考え方で、更新した内容は必要ときだけ下方に伝搬されることで効率的な運用が保証されます。

次の図は $\text{sum}_a(a,b)$ の値を計算したときに、ツリーがどのように変化するかを示したものである。矩形は値が変化するノードを示します。これは、遅延更新が下方に伝搬されるためである。



遅延更新の値はマージ、つまりすでに遅延更新を持つノードに別の遅延更新が割り当てられた場合の計算が必要なことに注意してください。クエリが和である場合は更新 z_1 と z_2 の組み合わせは $z_1 + z_2$ に対応するので、遅延更新を組み合わせるのは簡単ですが、演算の種類によりこの計算は変わります。

多項式の更新 - Polynomial updates

遅延更新は一般化でき、以下の形式の多項式を使用して区間を更新することが可能です。

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0.$$

この場合、 $[a, b]$ の i の位置の値の更新は $p(i-a)$ です。例えば多項式 $p(u) = u + 1$ を $[a, b]$ に追加すると、位置 a の値は 1 増加し、位置 $a+1$ の値は 2 増加し、などと処理されます。

多項式更新をサポートするためには k を多項式の次数として各ノードに $k+2$ 個の値が割り当てられます。値 s は区間内の要素の和であり値 z_0, z_1, \dots, z_k は遅延更新に対応する多項式の係数になります。

ここで、区間 $[x, y]$ の値の合計は、次の通りになります。

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0.$$

このような和の値は和の公式を使って効率的に計算することができます。例えば、 z_0 という項は和 $(y-x+1)z_0$ に対応し、 $z_1 u$ という項は次に対応します。

$$z_1(0+1+\dots+y-x) = z_1 \frac{(y-x)(y-x+1)}{2}.$$

更新を木に伝播する際、各区間 $[x, y]$ において $u = 0, 1, \dots, y-x$ の値を計算するため、 $p(u)$ のインデックスが変わりますが、 $p'(u) = p(u+h)$ は $p(u)$ と同次の多項式であるので問題にはなりません。例えば、 $p(u) = t_2 u^2 + t_1 u - t_0$ とすれば、以下のようになります。

$$p'(u) = t_2(u+h)^2 + t_1(u+h) - t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h - t_0.$$

28.2 動的セグメントツリー - Dynamic trees

通常セグメントツリーは静的に配列を確保するために各ノードの配列内での位置が固定されツリーには一定量のメモリが必要となります。**動的セグメントツリー - dynamic segment tree** ではアルゴリズム中に実際にアクセスされるノードに対してのみメモリを割り当て、大量のメモリを節約することができます。

これは構造体として表現します。

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

value はノードの値, $[x, y]$ は対応する区間、left と right はその子のツリーを指します。

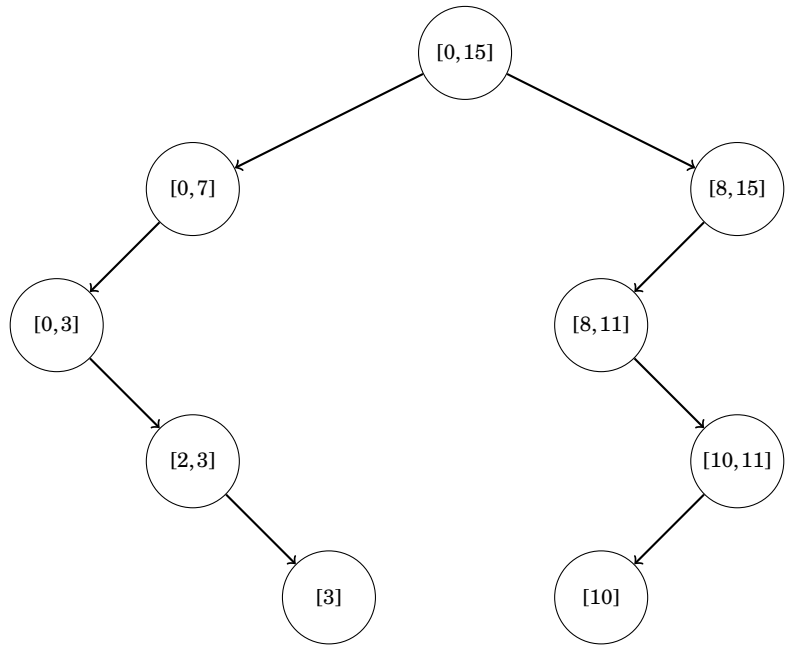
これを用いて次のようにノードを作成できます。

```
// create new node
node *x = new node(0, 0, 15);
// change value
x->value = 5;
```

スパースなセグメントツリー - Sparse segment trees

動的セグメントツリーは、ベースとなる配列が疎である場合、つまり、許容されるインデックスの区間 $[0, n-1]$ は大きい、ほとんどの配列値が 0 である場合に有効です。通常のセグメントツリーでは $O(n)$ のメモリを使用しますが動的セグメントツリーでは $O(k \log n)$ のメモリしか使用しません (k は実行される操作の数)。

スパースなセグメントツリー - sparse segment tree は最初は値がゼロのノード $[0, n-1]$ を 1 つだけ持ちます。これは、すべての配列の値がゼロであることを意味します。更新で必要があれば、新しいノードが動的にツリーに追加されます。たとえば $n = 16$ で位置 3 と 10 の要素が変更された場合にツリーには次のノードが含まれます。



ルートノードからリーフへのどのパスも $O(\log n)$ 個のノードを含むので、各操作でツリーに追加される新しいノードは最大 $O(\log n)$ 個です。したがって、 k 回の操作の後、木は最大 $O(k \log n)$ 個のノードを含むことになります。

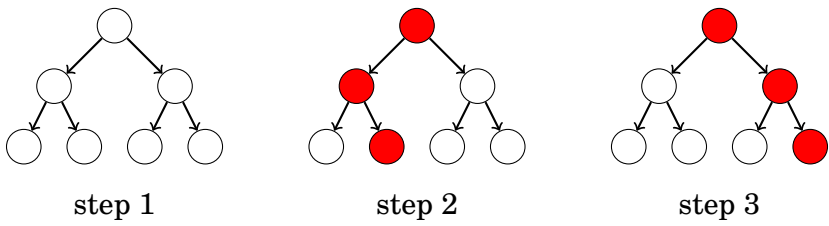
アルゴリズムの開始時に更新される要素がすべてわかっている場合は、動的なセグメントツリーは必要ありません。座標圧縮を行った通常のセグメントツリーを使用できるからです (9.4 章)。しかし、アルゴリズムの途中でインデックスが生成される場合は座標圧縮はできないのでこれらは有力なデータ構造の候補となります。

永続セグメントツリー - Persistent segment trees

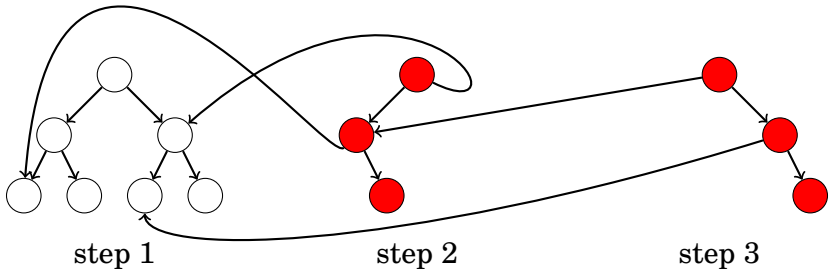
動的な実装を用いることで、ツリーの変更履歴 - *modification history* を保存する永続セグメントツリー - **persistent segment tree** を作成することもできます。この実装では、すべてのバージョンのツリーに効率的にアクセスすることができます。

修正履歴がある場合、各ツリーの完全な構造が保存されているため、通常のセグメントツリーのように、以前のツリーに対して問い合わせを行うことができます。また、以前の木を元に新しい木を作成し、独立して修正することもできます。

以下のような一連の更新を考えてみます。赤色のノードが変更されその他のノードはそのままだとしましょう。



各更新の後に木のほとんどのノードは同じままであるため、修正履歴を保存する効率的な方法は履歴の各木を新しいノードと以前のツリーのサブツリーの組み合わせで作ることです。この例では修正履歴は以下のように保存できます。



対応するルートノードから始まるポインタを辿ることで、以前の各木の構造を再構築することができます。各操作が木に追加する新しいノードは $O(\log n)$ だけなので、ツリーの全修正履歴を保存することが可能です。

28.3 データ構造 - Data structures

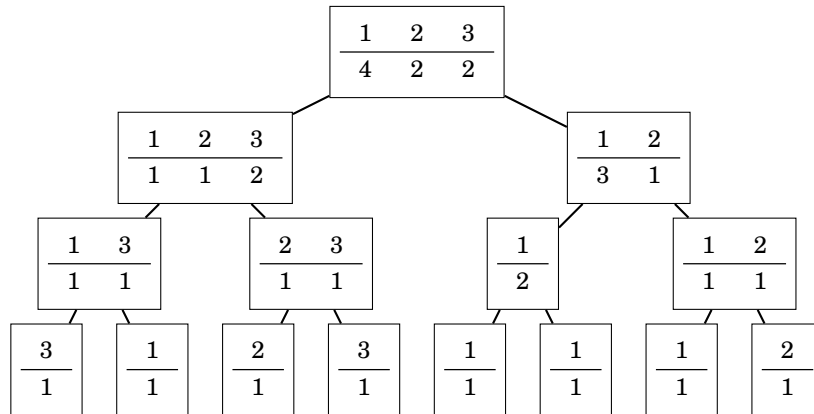
セグメントツリーのノードには、単一の値の代わりに、対応する区間に関する情報を保持する**データ構造 - data structures** を含めることもできます。このようなツリーでは、操作は $O(f(n)\log n)$ 時間を要します。ここで、 $f(n)$ は操作で 1 つのノードを処理するのに必要な時間です。

例として、”ある要素 x が $[a, b]$ の区間に何回出現するか?” というクエリをサポートするセグメントツリーを考えてみます。例えば、要素 1 が次の区間に 3 回現れるとします。

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

このような問い合わせをサポートするために、各ノードにデータ構造を割り当てたセグメントツリーを構築して任意の要素 x が対応する区間に何回現れるかをクエリできます。この木を用いて区間に属するノードからの結果をマージすることで問い合わせに対する答えが分かります。

以下のセグメントツリーは上記の配列に対応します。



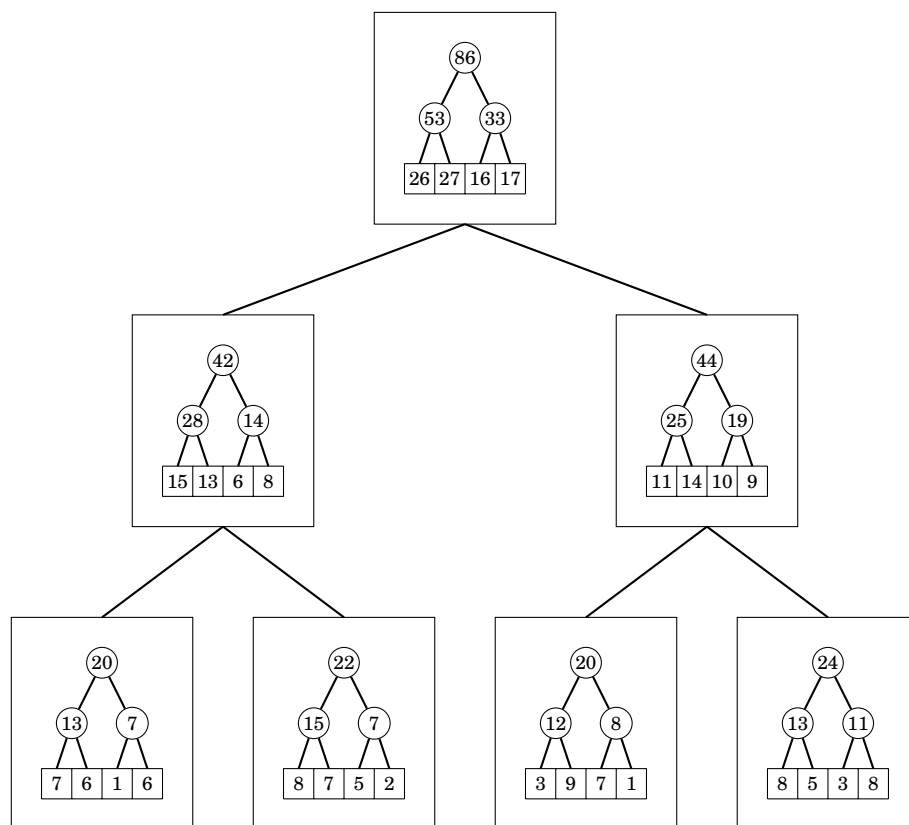
各ノードがマップ構造を含むようにツリーを構築すればよいです。各ノードの処理に必要な時間は $O(\log n)$ なのでクエリの総時間複雑度は $O(\log^2 n)$ となります。木は $O(\log n)$ のレベルがあり各レベルは $O(n)$ 個の要素を含むので $O(\log n)$ のメモリ使用量となります。

28.4 2次元セグメントツリー - Two-dimensionality

2次元セグメントツリー - two-dimensional segment tree は、2次元配列の矩形部分配列に関する問い合わせをサポートします。大きな木が配列の行に対応し、各ノードには列に対応する小さな木を含むような構造です。

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

このような2次元の配列があったとき、任意のサブアレイの和は以下のセグメントツリーから計算することができます。



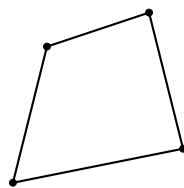
2次元のセグメント木の演算には、 $O(\log^2 n)$ の時間がかかります。これは、大きな木とそれぞれの小さな木が $O(\log n)$ のレベルで構成されているためです。各小さな木は $O(n)$ 個の値を含むので、木は $O(n^2)$ のメモリを必要とします。

第 29 章

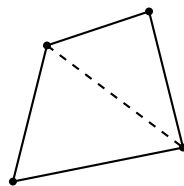
幾何学 - Geometry

幾何学問題は時に複雑で、問題の解を合理的に求めて特殊なケースの数が少なくなるようにアプローチする方法を見つけることはしばしば困難です。

例えば四辺形（4つの頂点を持つ多角形）の頂点が与えられて面積を計算する問題を考えてみましょう。この問題の入力として次のようなものが考えられます。



この問題へのアプローチの1つは対向する2つの頂点間の直線で2つの三角形に分割することです。

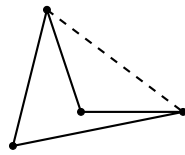


あとは、三角形の面積を合計すればよく三角形の面積は **Heron's formula** - ヘロンの公式で計算できます。

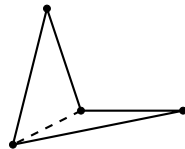
$$\sqrt{s(s-a)(s-b)(s-c)},$$

ここで、 a, b, c は三角形の各辺の長さで、 $s = (a + b + c)/2$ とします。

これはこれでよいアプローチなのですがひとつ落とし穴があります。四角形をどうやって三角形に分割するのか。実は任意の2つの対向する頂点を選べない場合があります。例えば、次のような場合に破線の分割線は四辺形の外に出てしまいます。



ただ、もう一つの線引きをすれば有効です。



これは人間にとってはどの線が正しいかは明らかなのですが、コンピュータにとっては難しい状況です。しかし、プログラマーにとってより便利な別の方法を用いて問題を解くことができます。

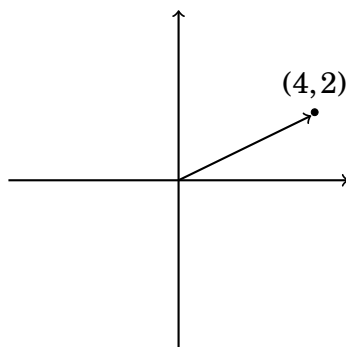
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

とすることによって四角形の面積が求められます。 $(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$ はそれぞれ4点の x, y 座標です。この公式は実装が簡単で特別なケースがありません。さらにすべての多角形に一般化することが可能です。

29.1 複素数 - Complex numbers

複素数 - complex number とは、 $x + yi$ の形の数で、 i は $i = \sqrt{-1}$ となる**虚数 - imaginary unit** です。複素数の幾何学的な解釈としては、2次元の点 (x, y) 、あるいは、原点から点 (x, y) へのベクトルを表すとします。

例えば $4 + 2i$ は次のような点とベクトルに対応します。



C++ の複素数クラス `complex` は幾何学的な問題を解くときに便利です。このクラスを使って点やベクトルを複素数として表現することができますし幾何学で役立つツールも含まれています。

以下のコードで、`C` は座標の型、`P` は点またはベクトルの型です。また、このコー

ドでは x 座標と y 座標を参照するために使用できるマクロ X と Y を定義しています。

```
typedef long long C;
typedef complex<C> P;
#define X real()
#define Y imag()
```

例えば、次のコードは、点 $p = (4, 2)$ を定義し、その x と y の座標を表示するものです。

```
P p = {4, 2};
cout << p.X << " " << p.Y << "\n"; // 4 2
```

次のコードはベクトル $v = (3, 1)$ と $u = (2, 2)$ を定義して和 $s = v + u$ を計算するものです。

```
P v = {3, 1};
P u = {2, 2};
P s = v + u;
cout << s.X << " " << s.Y << "\n"; // 5 3
```

実際には `long long` か `long double` (実数) が適切な座標型であることが多いはずです。整数を使った計算は正確なので、実数が求められない限りは整数を使うべきです。実数が必要な場合は、数値の比較の際に、推定誤差を考慮する必要があります。実数 a と b が等しいかどうかを確認する安全な方法は、 $|a - b| < \epsilon$ (訳註: `eps` で表現されることが多い) で比較することです。 ϵ は小さな数で例えば $\epsilon = 10^{-9}$ などです。

関数 - Functions

以下の例では型を `long double` とします。

`abs(v)` は、 $v = (x, y)$ の長さ $|v|$ を $\sqrt{x^2 + y^2}$ を用いて計算します。この関数は、距離の計算にも使うことができます。点 (x_1, y_1) と (x_2, y_2) の間の距離は、ベクトルの長さ $(x_2 - x_1, y_2 - y_1)$ に等しくなります。

次のコードは、点 $(4, 2)$ と $(3, -1)$ の間の距離を計算する例です。

```
P a = {4, 2};
P b = {3, -1};
cout << abs(b-a) << "\n"; // 3.16228
```

$\arg(v)$ はベクトル $v = (x, y)$ の x 軸に対する角度を計算します。この関数は角度をラジアン単位で与え、ここで r ラジアンとは $180r/\pi$ 度に相当します。真右を指すベクトルの角度は 0 であって角度は時計回りに減少して反時計回りに増加します。

$\text{polar}(s, a)$ は長さが s で、角度 a を指すベクトルを構成します。ベクトルの性質として長さが 1 で角度 a を持つベクトルと掛け合わせるによって、角度 a だけ回転させることができることに注意します。

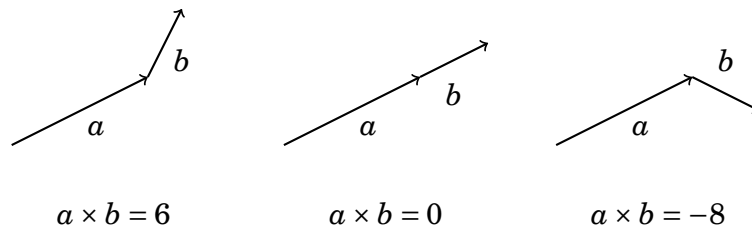
次のコードはベクトル $(4, 2)$ の角度を計算し、反時計回りに $1/2$ ラジアン回転させた後、再度角度を表示します。

```
P v = {4, 2};
cout << arg(v) << "\n"; // 0.463648
v *= polar(1.0, 0.5);
cout << arg(v) << "\n"; // 0.963648
```

29.2 点と線 - Points and lines

ベクトル $a = (x_1, y_1)$ と $b = (x_2, y_2)$ の外積 (ベクトル積) - **cross product** である $a \times b$ は、 $x_1 y_2 - x_2 y_1$ で求められます。外積は、 a に対して b が左回り (正の値) か直線 (ゼロ) か右回り (負の値) かを教えてください。

次の図は、上記のケースを説明したものです。



最初のケースでは $a = (4, 2)$ と $b = (1, 2)$ で、次のコードは `complex` クラスの機能を使い外積を計算します。

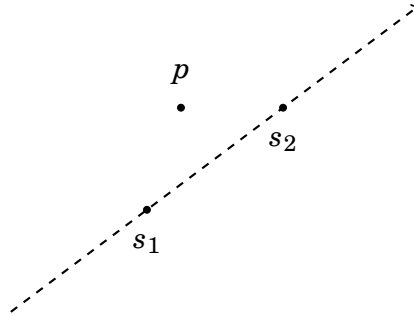
```
P a = {4, 2};
P b = {1, 2};
C p = (conj(a)*b).Y; // 6
```

関数 `conj` (訳註: 共役複素数を得る関数) がベクトルの y 座標を反転するので、ベクトル $(x_1, -y_1)$ と (x_2, y_2) となり、これを掛け合わせるので $x_1 y_2 - x_2 y_1$ となり結果は 6 になります。

点の位置 - Point location

外積はある点が直線の左側にあるか右側にあるかを知るのに利用できます。まず、線が点 s_1 と s_2 を通るとします。 s_1 側から s_2 を見て、その点を p 考えます。(訳註：どの点から見るかで左右は変わるので s_1 からとします)

例えば、次の図では、 p は左側にあります。

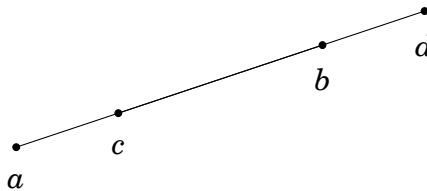


外積 $(p - s_1) \times (p - s_2)$ は点 p の位置を示します。結果が正の場合 p は左側に位置し、負の場合 p は右側に位置し、0 であれば点 s_1, s_2, p は同じ線上にあります。

線分の交差 - Line segment intersection

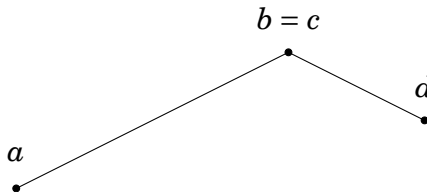
次に、2つの線分 ab と cd が同じあるいは交差するかを考えていきます。

Case 1: 線分が同じ線上にあり、互いに重なり合っている場合はこの場合、交点は無限に存在するといえます。例えば、次の図では、 c と b の間の点はすべて交点です。



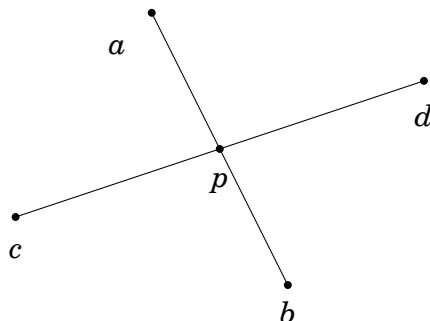
外積を用いるとすべての点が同一線上にあるかどうかを、確認することができます。点を並べ替えて、線分同士が重なっているかどうかを確認することができます。

Case 2: 線分が共通の頂点を持ちそれが唯一の交点となる場合もあります。次の図では交点は $b = c$ です。



交点の可能性は $a = c$ 、 $a = d$ 、 $b = c$ 、 $b = d$ の4つだけなので判定は簡単です。

Case 3: どの点でもない交差点を持つ場合があります。次の図では点 p が交点となります。



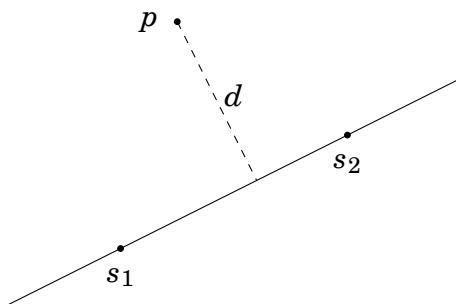
点 c と d が直線 ab の異なる側にいる時、線分は正確に交差します。これを確認するには外積を使えばよいです。

線分から点への距離 - Point distance from a line

また、外積の特徴として、三角形の面積を計算式で求めることができます。

$$\frac{|(a - c) \times (b - c)|}{2},$$

a, b, c は三角形の頂点の座標です。これを用いてある点と直線の最短距離を計算する公式を導くことができます。次の図において点 p と点 s_1 と s_2 で定義される直線との最短距離を d としましょう。

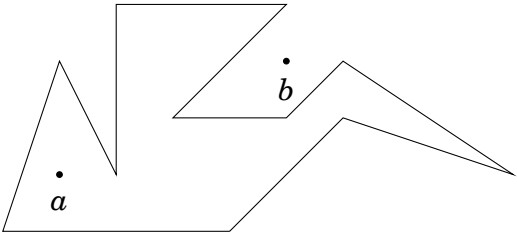


s_1, s_2, p を頂点とする三角形の面積は $\frac{1}{2}|s_2 - s_1|d$ あるいは $\frac{1}{2}((s_1 - p) \times (s_2 - p))$ の2つの方法で計算することができます。このため、最短の距離は以下の通りとなります。

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

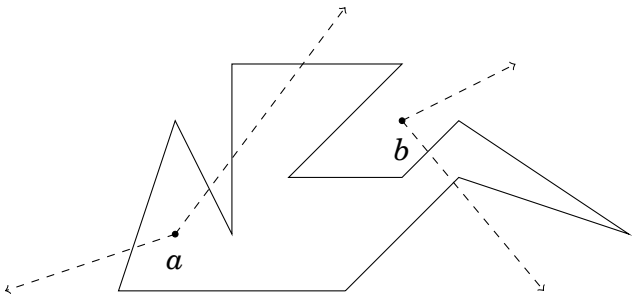
ポリゴン内の点 — Point inside a polygon

点が多角形の内側にあるか外側にあるかを判定する問題を考えます。例えば、次の図で点 a は多角形の内側にあり、点 b は多角形の外側にあるというとしましょう。



この問題の解決には興味深い方法が使えます。点から任意の方向に光線を送り（つまり、点から十分に長い線を書く）、それが多角形の線に触れる回数を計算します。その回数が奇数なら、その点は多角形の内側にあり、偶数なら多角形の外側にあることが知られています。

例を示します。



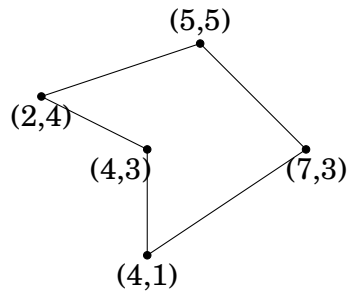
a からの光線は、ポリゴンと 1 回、3 回接するので、 a はポリゴンの内側にあることになります。これに対応して b からの光線はポリゴンの境界と 0 回と 2 回接するので、 b はポリゴンの外側にあることになります。

29.3 ポリゴンの面積 - Polygon area

多角形の面積を計算するための一般的な公式は**靴紐アルゴリズム - shoelace formula**と呼ばれ、以下のように示されます。

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

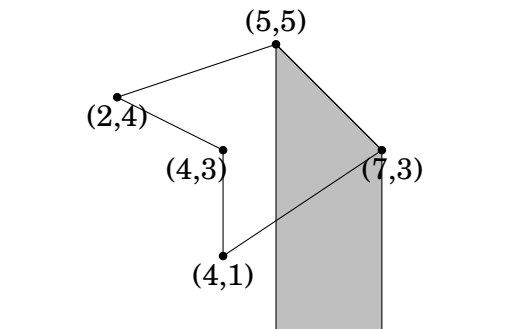
頂点は、 $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ であり p_i と p_{i+1} が多角形の境界で隣接して最初と最後の頂点が同じになる順序、つまり $p_1 = p_n$ とします。



このポリゴンの面積の面積は以下の通りです。

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

この考え方は y 軸を一辺に持ち、対向の一辺をポリゴンの辺とする台形を用いて面積を求めています。



このような台形の面積は、 p_i と p_{i+1} に注目した時、

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

となります。先ほどのルールに従い点は並んでいるので $x_{i+1} > x_i$ であるとき面積は加算すべきで、 $x_{i+1} < x_i$ であるとき面積は減算すべきです。

このようにポリゴンの面積を台形の面積で求めることができました。

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

ポリゴンの境界に沿って時計回りに走査するか反時計回りに走査するかによって、和の値が正または負になることがあるので、和の絶対値をとることに注意してください。

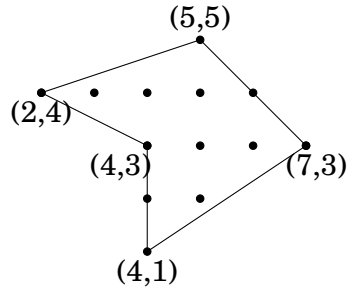
ピックの定理 - Pick's theorem

ピックの定理 - Pick's theorem は、多角形のすべての頂点が整数の座標を持つ場合に、多角形の面積を計算する方法を提供します。ピックの定理によれば多角形

の面積は次で示すことができます。

$$a + b/2 - 1,$$

a はポリゴン内部の整数点の数で b はポリゴンの境界上の整数点の数です。



このポリゴンの面積は $6 + 7/2 - 1 = 17/2$ となります。

29.4 距離関数 - Distance functions

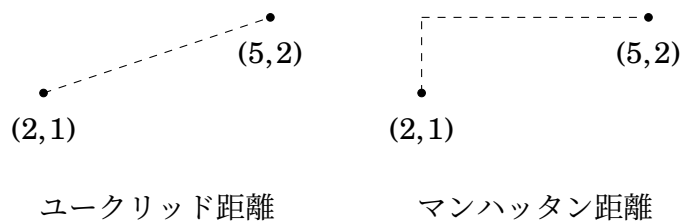
距離関数 - distance function は 2 点の距離を求める関数です。一般的な距離は**ユークリッド距離 - Euclidean distance** で、 (x_1, y_1) と (x_2, y_2) に対して以下のように定まります。

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

他には**マンハッタン距離 - Manhattan distance** も使われこれは以下のように示されます。

$$|x_1 - x_2| + |y_1 - y_2|.$$

次のように絵で考えてみます。



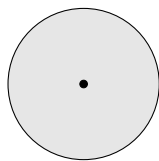
ユークリッド距離は、

$$\sqrt{(5-2)^2 + (2-1)^2} = \sqrt{10}$$

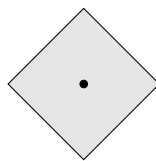
となり、マンハッタン距離は

$$|5-2| + |2-1| = 4.$$

となります。次の図は、ユークリッド距離とマンハッタン距離を用いた中心点から 1 以内の距離にある領域です。



Euclidean distance

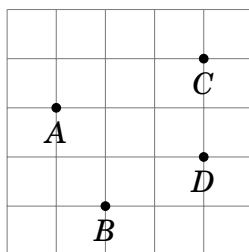


Manhattan distance

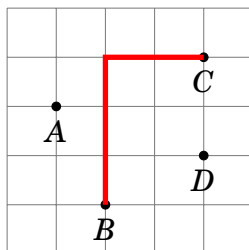
回転座標 - Rotating coordinates

ユークリッド距離の代わりにマンハッタン距離を使うと解きやすくなる問題もあります。例として2次元平面上の n 個の点を与えられ、任意の2点間の最大マンハッタン距離を計算する問題を考えてみます。

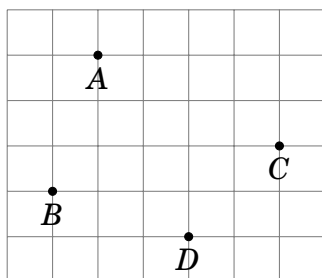
例えば、次のような点の集合があったとしましょう。



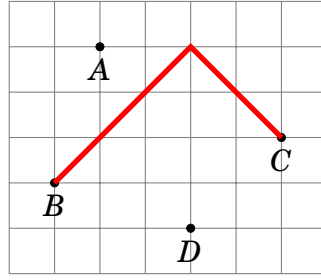
ここでのマンハッタン距離の最大値は B と C の5です。



マンハッタン距離を用いた便利なテクニックは点 (x, y) が $(x + y, y - x)$ になるように座標を45度回転させることです。上記の点を回転させた結果は以下のようになります。



最大の距離は以下のようになります。



2つの点 $p_1 = (x_1, y_1)$ と $p_2 = (x_2, y_2)$ を回転させた座標である $p'_1 = (x'_1, y'_1)$ と $p'_2 = (x'_2, y'_2)$ を考えます。 p_1 と p_2 のマンハッタン距離は2つの方法で表現できます。

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

例えば、 $p_1 = (1, 0)$ と $p_2 = (3, 3)$ の場合に回転した座標は $p'_1 = (1, -1)$ と $p'_2 = (6, 0)$ であり、このマンハッタン距離は次の通りです。

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

回転座標は x と y の座標を別々に考えることができるため、マンハッタン距離の計算は簡単に行うことができます。2点間のマンハッタン距離を最大にするには回転座標の値が最大となる2点を見つければよいです。

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|).$$

これは簡単に求められます。回転した座標の水平または垂直方向の差のどちらかが最大であればよいからです。

第 30 章

掃引線アルゴリズム - Sweep line algorithms

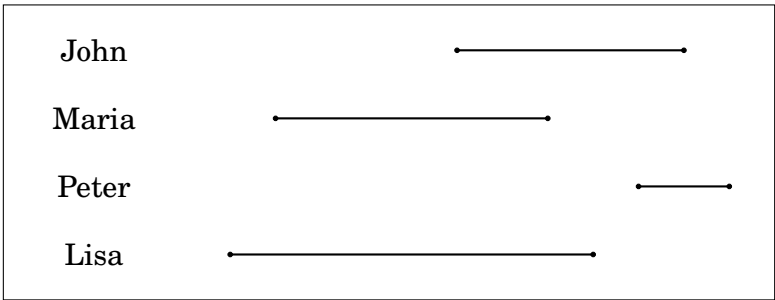
幾何学的な問題のいくつかは**掃引線 - sweep line** を用いたアルゴリズムで解くことができます。これは問題を平面上の点に対応するイベントの集合として表現するアイデアです。イベントはその x または y に従って昇順に処理します。

例えば従業員数 n 人の会社があり、各従業員のある日の出社・退社時刻がわかっているとします。我々の仕事は、同時にオフィスにいた従業員の最大人数を計算することです。

この問題は各従業員に出社時間と退社時間に対応する 2 つのイベントを割り当てるようにして解決できます。イベントを分類した後そのイベントを調べてオフィスにいる人数を把握します。

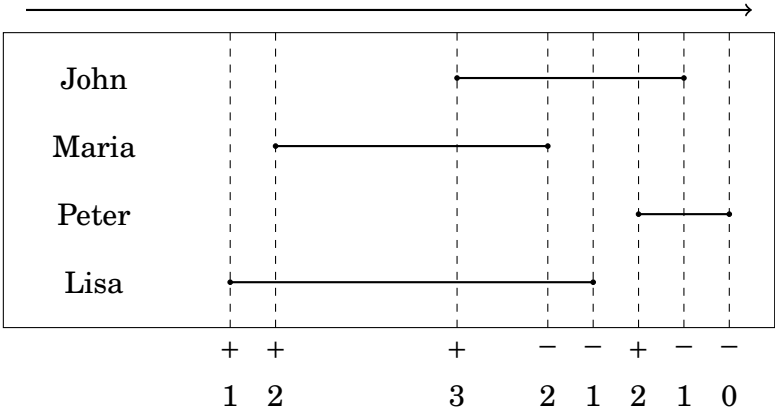
名前	出社時間	退社時間
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

これは次のようなイベントです。



このとき、左から右へイベントをたどってカウンタを管理します。出社イベントの時はカウンタをインクリメントして、退社イベントの時はカウンタをデクリメントします。こうすると答えはその間の最大値になります。

この例では次のように処理が行われます。

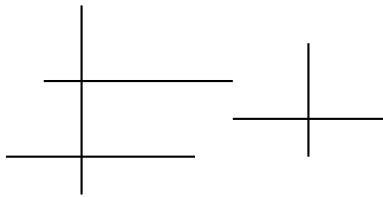


各記号はカウンターの値が増加するか減少するかを示し、カウンターの値は記号の下に示すようになります。カウンターの最大値は、ジョンが到着してからマリアが帰るまでの間の3です。

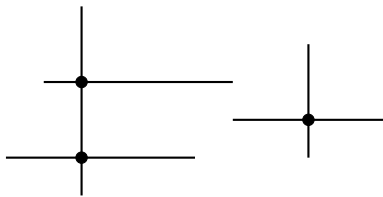
イベントのソートに $O(n \log n)$ の時間がかかり、残りのアルゴリズムに $O(n)$ の時間がかかるため、このアルゴリズムの実行時間は $O(n \log n)$ である。

30.1 交差点 - Intersection points

各々が水平または垂直な n 本の線分の集合があるとき、交点の総数を数える問題を考えましょう。



この場合の交差点は3つです。

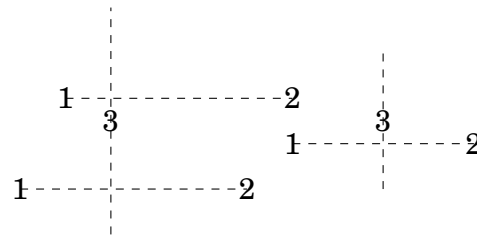


この問題を $O(n^2)$ 時間で解くのは簡単です。全ての線分の組を調べて、それらが

交差しているかどうかをチェックします。しかし、掃引線アルゴリズムと区間クエリデータ構造を用いれば $O(n \log n)$ の時間で解けます。線分の端点を左から右へ処理しながら次の 3 種類の事象に注目するというものです。

- (1) 水平線の開始
- (2) 水平線の終了
- (3) 垂直線

先ほどの例では次のようなイベントの対応になります。



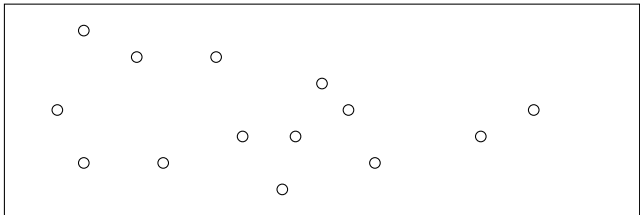
イベントを左から右へみていき、アクティブな水平線が存在する y 座標のセットを保持するデータ構造を持ちます。

イベント 1 では、セグメントの y 座標をセットに追加します。イベント 2 では、その y 座標をセットから削除します。イベント 3 では、交点を計算します。点 y_1 と y_2 の間に垂直セグメントがある場合、 y 座標が y_1 と y_2 の間にあるアクティブな水平セグメントの数を数えてこの数を交点の総数に加えます。

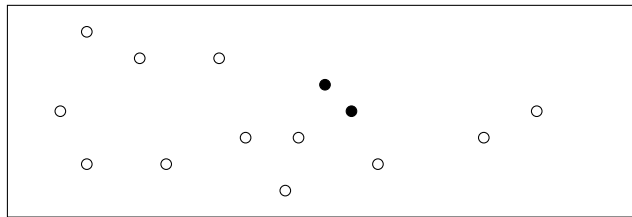
水平線の y 座標を格納するために BIT または セグメントツリーを使用した場合によっては座標圧縮を行います。このような構造を使用する場合は各イベントの処理に $O(\log n)$ の時間がかかるため、アルゴリズムの総実行時間は $O(n \log n)$ となります。

30.2 近接ペア問題 - Closest pair problem

n 個の点の集合が与えられ、ユークリッド距離が最小となる 2 点を見つける問題を考えます。例えば、点が



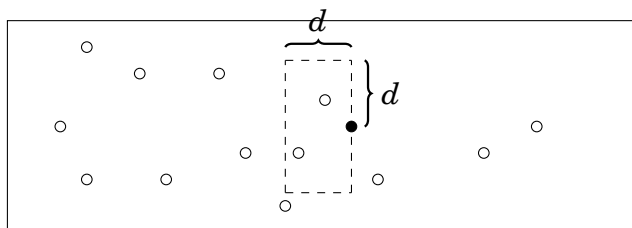
のとき、以下の2点を見つけます。



これも掃引線アルゴリズムを用いると $O(n \log n)$ の時間で解ける問題の一例です*¹。左から右へと点进行处理しながら、これまでに見た2点間の最小距離である値 d を保持します。このとき各点で、左側に最も近い点を探します。その距離が d より小さい場合、それが新しい最小距離となり d の値を更新します。

現在の点が (x, y) として、左側に d 以下の距離に点がある場合、その点の x 座標は $[x-d, x]$ の間でなければならず、 y 座標は $[y-d, y+d]$ の間でなければなりません。これらの区間に位置する点のみを考慮すれば十分であり、このアルゴリズムは効率的に動作します。

例えば、以下の図において破線で示した領域は現在の値から d 以内の距離に存在する点を表すことになります。



このアルゴリズムが効率的に動くのは各領域には常に $O(1)$ 個の点しか含まれないという事実に基づいています。 x 座標が $[x-d, x]$ の間にある点の集合を y 座標に従って昇順に保持することにより $O(\log n)$ 時間でそれらの点を走査することができるのです。

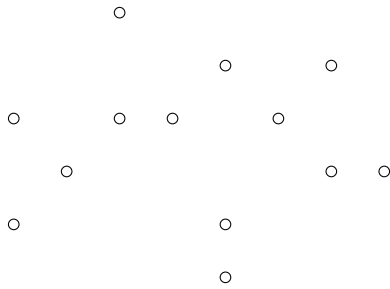
このアルゴリズムの時間計算量は $O(n \log n)$ です。 n の各点について左に最も近い点を $O(\log n)$ 時間で求められます。

30.3 凸包問題 - Convex hull problem

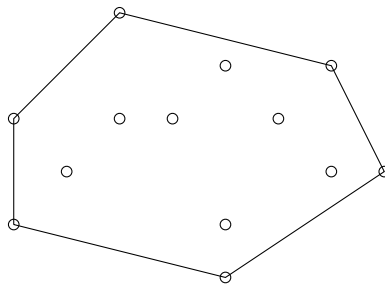
凸包 - convex hull とは与えられた集合のすべての点を含む最小の凸の多角形のことです。凸とは多角形の任意の2つの頂点を結ぶ線分が完全にある多角形の内

*¹ Besides this approach, there is also an $O(n \log n)$ time divide-and-conquer algorithm [56] that divides the points into two sets and recursively solves the problem for both sets.

部にあることを意味します。
 例えば以下の点を考えます。

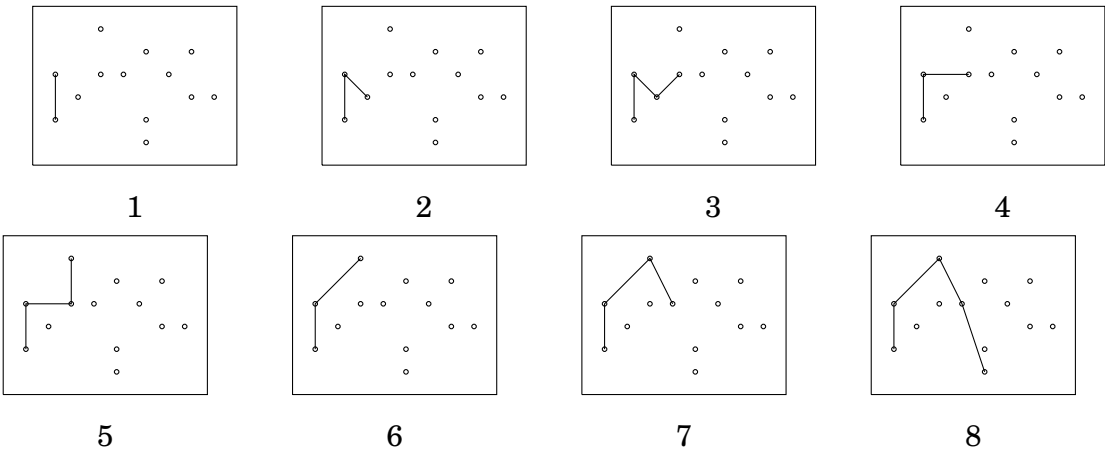


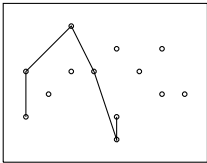
この時の凸包は以下のとおりです。



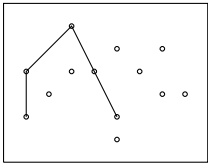
アンドリューのアルゴリズム - Andrew's algorithm [3] は、点群に対する凸包を $O(n \log n)$ 時間で構成するアルゴリズムです。このアルゴリズムでは、まず左端と右端の点を求め、凸包を上包と下包の 2 つに分けます。

最初は上部の構築に集中します。点を主に x 座標で、次に y 座標の順で並べ替えます。その後、各ポイントを凸包に追加していきます。点を追加した後は、常に凸包の最後の線分が左に曲がらないようにします。左に曲がっているならば最後から 2 つ目の点を凸包から取り除いていきます。以下の図は **Andrew** のアルゴリズムがどのように動作するかを示しています。

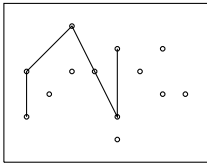




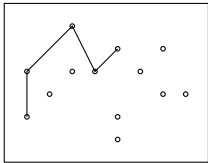
9



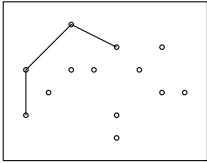
10



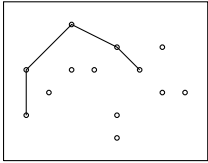
11



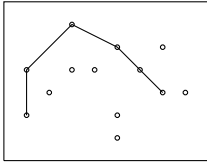
12



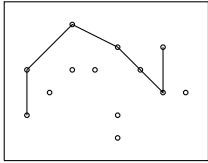
13



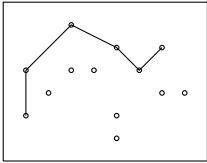
14



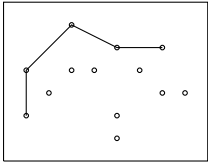
15



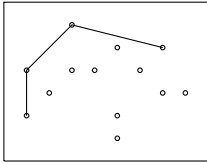
16



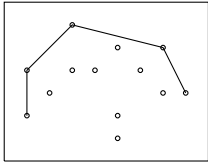
17



18



19



20

参考文献

- [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.

索引

- 2-SAT 問題 - 2SAT problem, 186
- 2SUM 問題 - 2SUM problem, 90
- 2 ポインタ - two pointers method, 89
- 2 次元セグメントツリー - two-dimensional segment tree, 301
- 3SAT problem, 188
- 3SUM problem, 91
- amortized analysis, 89
- AND 演算 - and operation, 110
- BIT - binary indexed tree, 99
- child, 156
- Chinese remainder theorem, 234
- complex, 304
- coprime, 230
- cycle, 179
- Dijkstra's algorithm, 146
- Diophantine equation, 233
- divisibility, 225
- divisor, 225
- Euclid's algorithm, 229
- Euclid's formula, 235
- Euler tour technique, 195
- Euler's theorem, 231
- Euler's totient function, 230
- extended Euclid's algorithm, 233
- factor, 225
- Fermat's theorem, 231
- Fibonacci number, 235
- Floyd – Warshall algorithm, 150
- Freivalds' algorithm, 265
- functional graph, 178
- Goldbach's conjecture, 227
- greatest common divisor, 229
- Grundy ゲーム - Grundy's game, 274
- Grundy 数 - Grundy number, 271
- harmonic sum, 229
- in-order, 162
- Kadane のアルゴリズム - Kadane's algorithm, 26
- Kosaraju's algorithm, 184
- Kruskal's algorithm, 164
- k 番目に小さい数 (Order statistics), 264
- Laplacean matrix, 256
- leaf, 155
- least common multiple, 229
- Legendre's conjecture, 227

- map, 45
- maximum spanning tree, 164
- MEX - mex function, 271
- minimum spanning tree, 163
- misère game, 270
- Mo's algorithm, 290
- modular arithmetic, 230
- modular inverse, 231
- negative cycle, 146
- next_permutation, 58
- Nim - nim game, 269
- nim sum, 269
- NOT 演算 - not operation, 111
- NP 困難 - NP-hard problem, 23
- number theory, 225
- OR 演算 - or operation, 111
- ペア - pair, 35
- parent, 156
- post-order, 162
- pre-order, 162
- prime decomposition, 225
- quickselect, 264
- quicksort, 264
- random_shuffle, 46
- reverse, 46
- set, 43
- sieve of Eratosthenes, 228
- sort, 46
- spanning tree, 163
- SPFA algorithm, 146
- square root algorithm, 285
- string, 42
- tree traversal array, 190
- タプル - tuple, 35
- typedef, 9
- twin prime, 227
- union-find structure, 167
- Wilson's theorem, 235
- XOR 演算 - xor operation, 111
- Z-アルゴリズム - Z-algorithm, 281
- Zeckendorf's theorem, 235
- Z 配列 - Z-array, 281
- アルファベット - alphabet, 275
- アンドリューのアルゴリズム - Andrew's algorithm, 319
- イテレータ - iterator, 46
- ウォーンスドルフの法則 - Warnsdorf's rule, 208
- エッジリスト - edge list, 132
- エドモンズ・カープのアルゴリズム - Edmonds - Karp algorithm, 213
- オイラー路 - Eulerian path, 201
- オイラー閉路 - Eulerian circuit, 202
- オレの定理 - Ore's theorem, 206
- カウントソート - counting sort, 33
- カタラン数 - Catalan number, 241
- カット - cut, 210
- キュー - queue, 50
- キルヒホッフの定理 - Kirchhoff's theorem, 256
- クイーン問題 - queen problem, 58
- グラフ - Graph, 125
- ケイリーの公式 - Cayley's formula, 246

- ケーニヒの定理 - Knig's theorem, 218
- コードワード - codeword, 71
- サクセスグラフ - successor graph, 178
- サフィックス - suffix, 276
- スケーリングアルゴリズム - scaling algorithm, 213
- スタック - stack, 50
- スパースなセグメントツリー - sparse segment tree, 298
- スパーステーブル - sparse table, 97
- スプレイグ・グランディの定理 - Sprague - Grundy theorem, 271
- スライディングウィンドウ - sliding window, 93
- セグメントツリー - segment tree, 102, 293
- sort, 34
- ソートアルゴリズム - sorting, 29
- ダイクストラアルゴリズム, 177
- ディラックの定理 - Dirac's theorem, 206
- ディルワースの定理 - Dilworth's theorem, 222
- デック - deque, 49
- デ・ブルーイェン配列 - De Bruijn sequence, 207
- データ圧縮 - data compression, 71
- データ構造 - data structure, 41
- トポロジカルソート - topological sorting, 173
- トライ - trie, 276
- ナイトツアー - knight's tour, 207
- ナップザック問題 - knapsack, 83
- ノード - node, 125
- ハッシュ - hashing, 278
- ハッシュ値 - hash value, 278
- ハフマン符号化 - Huffman coding, 72
- ハミルトン路 - Hamiltonian path, 205
- ハミルトン閉路 - Hamiltonian circuit, 205
- ハミング距離 - Hamming distance, 115
- バイナリコード - binary code, 71
- バイナリサーチ, 36
- バックトラッキング - backtracking, 58
- バブルソート - bubble sort, 30
- バーンサイドの補題 - Burnside's lemma, 245
- パスカルの三角形 - Pascal's triangle, 239
- パターンマッチング - pattern matching, 275
- ヒューリスティック - heuristic, 208
- ヒープ - heap, 50
- ヒールホルツァーのアルゴリズム - Hierholzer's algorithm, 203
- ビットシフト - bit shift, 111
- ビットセット - bitset, 48
- ビット表現 - bit representation, 109
- ビネの公式 - Binet's formula, 15
- ピタゴラスの定理 - Pythagorean triple, 235
- ピックの定理 - Pick's theorem, 310
- ピリオド - period, 276
- ファウルハーバーの公式 - Faulhaber's formula, 11

- フィボナッチ数 - Fibonacci number, 15, 252
- フェニック木 - Fenwick tree, 99
- フォード・ファルカーソンのアルゴリズム - Ford - Fulkerson algorithm, 211
- フロイドの循環検出アルゴリズム - Floyd's algorithm, 180
- フロー - flow, 209
- プリムのアルゴリズム - Prim's algorithm, 169
- プリューファー列 - Prüfer code, 246
- プレフィックス - prefix, 276
- プログラミング言語, 3
- ヘロンの公式 - Heron's formula, 303
- ベクトル - vector, 249, 304
- ホールの定理 - Hall's theorem, 217
- ボーダー - border, 276
- マクロ - macro, 9
- マッチング - matching, 216
- マルコフ連鎖 - Markov chain, 263
- マンハッタン距離 - Manhattan distance, 311
- マージソート - merge sort, 31
- メモ化 - memoization, 77
- モジュロ演算 - modular arithmetic, 7
- モンテカルロ法 (Monte Carlo algorithm), 264
- ユークリッド距離 - Euclidean distance, 311
- ラグランジュの定理 - Lagrange's theorem, 235
- ラスベガス法 (Las Vegas algorithm), 264
- レーベンシュタイン距離 - Levenshtein distance, 85
- 一様分布 - uniform distribution, 262
- 三乗アルゴリズム - cubic algorithm, 23
- 乱択, 264
- 二乗アルゴリズム - quadratic algorithm, 22
- 二分探索 - binary search, 36
- 二分木 - binary tree, 161
- 二部グラフ - bipartite graph, 128, 141
- 二項係数 - binomial coefficient, 238
- 二項分布 - binomial distribution, 262
- 交差点 - intersection point, 316
- 余因子 - cofactor, 251
- 優先度付きキュー - priority queue, 50
- 入出力, 5
- 入次数 - indegree, 128
- 全体集合 - universal set, 13
- 全域木 - spanning tree, 256
- 出次数 - outdegree, 128
- 分布 - distribution, 262
- 剰余 - remainder, 7
- 動的セグメントツリー - dynamic segment tree, 297
- 動的計画法 - dynamic programming, 75
- 動的配列 - dynamic array, 41
- 勝ち状態 - winning state, 267
- 包除原理 - inclusion-exclusion, 243
- 区間クエリ - range query, 95
- 半分全列挙 - meet in the middle, 63
- 単位行列 - identity matrix, 250
- 単純グラフ - simple graph, 129
- 反鎖 - antichain, 222

- 含意 - implication, 14
- 和のクエリ - sum query, 95
- 回転 - rotation, 276
- 外積 - cross product, 306
- 多項係数 - multinomial coefficient, 240
- 多項式アルゴリズム - polynomial algorithm, 23
- 多項式ハッシュ - polynomial hashing, 278
- 完全グラフ - complete graph, 128
- 完全マッチング - perfect matching, 217
- 完全数 - perfect number, 226
- 完全順列 - derangement, 244
- 定数時間 - constant-time algorithm, 22
- 定数要素 - constant factor, 24
- 対数 - logarithm, 15
- 対数アルゴリズム - logarithmic algorithm, 22
- 対等 - equivalence, 14
- 差分配列 - difference array, 107
- 幅優先探索 - breadth-first search, 137
- 幾何分布 - geometric distribution, 262
- 幾何学 - geometry, 303
- 座標圧縮 - index compression, 106
- 強連結グラフ - strongly connected graph, 183
- 強連結成分 - strongly connected component, 183
- 彩色 - coloring, 128, 265
- 成分 - component, 126
- 成分グラフ - component graph, 183
- 括弧表現 - parenthesis expression, 241
- 掃引線 - sweep line, 315
- 整数 - integer, 6
- 文字列 - string, 275
- 文字列のハッシュ化 - string hashing, 278
- 時間計算量 - time complexity, 19
- 時間計算量の種類 - complexity classes, 22
- 最も近い小さな要素 - nearest smaller elements, 91
- 最大クエリ - maximum query, 95
- 最大マッチング - maximum matching, 216
- 最大流 - maximum flow, 209
- 最大独立集合 - maximum independent set, 219
- 最小カット - minimum cut, 210, 213
- 最小クエリ - minimum query, 95
- 最小スライディングウィンドウ - sliding window minimum, 93
- 最小共通祖先 (LCA) - lowest common ancestor, 193
- 最小点被覆 - minimum node cover, 218
- 最短経路, 143
- 最短経路 (Bellman – Ford), 143
- 最長増加部分列 - longest increasing subsequence, 81
- 有向グラフ - directed graph, 127
- 期待値 - expected value, 261
- 木, 155
- 木 - tree, 126

木に対するクエリ - tree query, 189
 条件付き確率 - conditional probability, 260
 根, 155
 根付き木, 155
 次数 - degree, 127
 正則グラフ - regular graph, 128
 正方向行列 - square matrix, 249
 比較演算子 - comparison operator, 35
 比較関数 - comparison function, 36
 永続セグメントツリー - persistent segment tree, 299
 浮動小数点数 - floating point number, 8
 深さ優先探索 - depth-first search, 135

 点 - point, 304
 点被覆 - node cover, 218
 独立 - independence, 260
 独立集合 - independent set, 219
 直径, 157
 確率 - probability, 257
 確率変数 - random variable, 261
 祖先 - ancestor, 189
 等差数列 - arithmetic progression, 11
 等比数列 - geometric progression, 12
 素数 - prime, 225
 累積和 - prefix sum array, 96
 組合わせ論 - combinatorics, 237
 経路 - path, 125
 線分の交差 - line segment intersection, 307
 線型回帰 - linear recurrence, 252
 線形アルゴリズム - linear algorithm,

編集距離 - edit distance, 85

 自然対数 - natural logarithm, 16
 行列 - matrix, 249
 行列の乗算の検証 - matrix multiplication, 265
 行列の累乗 - matrix power, 251
 行列同士の乗算 - matrix multiplication, 250
 行列式 - determinant, 251
 衝突 - collision, 279
 補集合 - complement, 13
 複素数 - complex number, 304
 誕生日のパラドックス - birthday paradox, 280
 調和級数 - harmonic sum, 12
 論理否定 - negation, 14
 論理和 - conjunction, 14
 論理式 - logic, 14
 論理積 - disjunction, 14
 論理述語 - predicate, 14
 負け状態 - losing state, 267
 貪欲アルゴリズム - greedy algorithm, 65
 距離関数 - distance function, 311
 転置 - inversion, 31
 転置行列 - transpose, 249
 辞書順 - lexicographical order, 276
 辺 - edge, 125
 辺被覆 - path cover, 219
 近接ペア - closest pair, 317

 逆行列 - inverse matrix, 252
 連結グラフ - connected graph, 126,

- 遅延セグメントツリー - lazy segment tree, 294
- 遅延伝搬 - lazy propagation, 294
- 部分文字列 - substring, 275
- 部分木, 156
- 部分配列 - subsequence, 275
- 部分配列の和の最大値 - maximum subarray sum, 24
- 部分集合 - subset, 13, 55
- 配列 - vector, 41
- 重み付きグラフ - weighted graph, 127
- 量化子 - quantifier, 14
- 閉路 - cycle, 126, 140, 173
- 閉路検出 - cycle detection, 179
- 階乗 - factorial, 15
- 隣接 - neighbor, 127
- 隣接リスト - adjacency list, 129
- 隣接行列 - adjacency matrix, 131
- 集合 - set, 13
- 集合和 - intersection, 13
- 集合差 - difference, 13
- 集合積 - union, 13
- 集合論 - set theory, 13
- 靴紐アルゴリズム - shoelace formula, 309
- 順列 - permutation, 57