

競技プログラマー ハンドブック

Competitive Programmer's Handbook

原著: Antti Laaksonen

Draft 2023 年 1 月 1 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 Contests and resources | 17 |
| 第 2 章 時間計算量 - Time complexity | 21 |
| 2.1 計算ルール - Calculation rules | 21 |
| 2.2 時間計算量の種類 - Complexity classes | 24 |
| 2.3 効率の見積もり - Estimating efficiency | 25 |
| 2.4 部分配列の和の最大値 - Maximum subarray sum | 26 |
| 第 3 章 ソート - Sorting | 31 |
| 3.1 ソートの理論 - Sorting theory | 31 |
| 3.2 C++ のソート - Sorting in C++ | 36 |
| 3.3 二分探索 (バイナリサーチ) - Binary search | 38 |
| 第 4 章 Data structures | 43 |
| 4.1 Dynamic arrays | 43 |
| 4.2 Set structures | 45 |
| 4.3 Map structures | 47 |
| 4.4 Iterators and ranges | 48 |

| | | |
|--------|---|-----|
| 4.5 | Other structures | 50 |
| 4.6 | Comparison to sorting | 54 |
| 第 5 章 | 全探索 - Complete search | 57 |
| 5.1 | 部分集合を全生成する - Generating subsets | 57 |
| 5.2 | 順列の生成 - Generating permutations | 59 |
| 5.3 | バックトラッキング - Backtracking | 60 |
| 5.4 | Pruning the search | 62 |
| 5.5 | Meet in the middle | 65 |
| 第 6 章 | Greedy algorithms | 67 |
| 6.1 | Coin problem | 67 |
| 6.2 | Scheduling | 69 |
| 6.3 | Tasks and deadlines | 71 |
| 6.4 | Minimizing sums | 72 |
| 6.5 | Data compression | 73 |
| 第 7 章 | Dynamic programming | 77 |
| 7.1 | Coin problem | 77 |
| 7.2 | Longest increasing subsequence | 83 |
| 7.3 | Paths in a grid | 84 |
| 7.4 | Knapsack problems | 86 |
| 7.5 | Edit distance | 87 |
| 7.6 | Counting tilings | 89 |
| 第 8 章 | ならし解析 - Amortized analysis | 91 |
| 8.1 | 2 ポインタ - Two pointers method | 91 |
| 8.2 | Nearest smaller elements | 93 |
| 8.3 | Sliding window minimum | 95 |
| 第 9 章 | Range queries | 97 |
| 9.1 | Static array queries | 98 |
| 9.2 | Binary indexed tree | 101 |
| 9.3 | Segment tree | 104 |
| 9.4 | Additional techniques | 108 |
| 第 10 章 | Bit manipulation | 111 |

| | | |
|--|---|------------|
| 10.1 | Bit representation | 111 |
| 10.2 | Bit operations | 113 |
| 10.3 | Representing sets | 115 |
| 10.4 | Bit optimizations | 117 |
| 10.5 | Dynamic programming | 119 |
| 第 II 部 グラフアルゴリズム - Graph algorithms | | 125 |
| 第 11 章 グラフの基礎知識 - Basics of graphs | | 127 |
| 11.1 | グラフの用語 - Graph terminology | 127 |
| 11.2 | グラフの表現方法 - Graph representation | 131 |
| 第 12 章 グラフ探索 - Graph traversal | | 137 |
| 12.1 | 深さ優先探索 - Depth-first search | 137 |
| 12.2 | 幅優先探索 - Breadth-first search | 139 |
| 12.3 | 応用 - Applications | 141 |
| 第 13 章 最短経路 | | 145 |
| 13.1 | 最短経路 (Bellman – Ford) | 145 |
| 13.2 | ダイクストラ法 - Dijkstra’s algorithm | 148 |
| 13.3 | ワーシャルフロイド法 - Floyd – Warshall algorithm | 152 |
| 第 14 章 木のアルゴリズム - Tree algorithms | | 157 |
| 14.1 | 木の走査 - Graph Traversal | 158 |
| 14.2 | 直径 - Diameter | 159 |
| 14.3 | 全ノードからの最長経路 - All longest paths | 162 |
| 14.4 | 二分木 - Binary trees | 164 |
| 第 15 章 全域木 - Spanning trees | | 167 |
| 15.1 | クラスカル法 - Kruskal’s algorithm | 168 |
| 15.2 | Union-find 構造 - Union-find structure | 171 |
| 15.3 | Prim’s algorithm | 173 |
| 第 16 章 有向グラフ - Directed graphs | | 177 |
| 16.1 | トポロジカルソート - Topological sorting | 177 |
| 16.2 | 動的計画法 - Dynamic programming | 180 |
| 16.3 | サクセスパス - Successor paths | 182 |

| | | |
|---------|---|-----|
| 16.4 | 閉路検出 - Cycle detection | 184 |
| 第 17 章 | 強連結 - Strong connectivity | 187 |
| 17.1 | Kosaraju's algorithm | 188 |
| 17.2 | 2-SAT 問題 - 2SAT problem | 190 |
| 第 18 章 | 木に対するクエリ - Tree queries | 193 |
| 18.1 | 祖先の検索 - Finding ancestors | 193 |
| 18.2 | 部分木とパス - Subtrees and paths | 194 |
| 18.3 | 最小共通祖先 (LCA) - Lowest common ancestor | 197 |
| 18.4 | オフラインのアルゴリズム - Offline algorithms | 200 |
| 第 19 章 | 経路と閉路 - Paths and circuits | 205 |
| 19.1 | オイラー経路 - Eulerian paths | 205 |
| 19.2 | ハミルトン路 - Hamiltonian paths | 210 |
| 19.3 | デ・ブルーイェン配列 - De Bruijn sequences | 211 |
| 19.4 | ナイトツアー - Knight's tours | 212 |
| 第 20 章 | フローとカット - Flows and cuts | 215 |
| 20.1 | フォード・ファルカーソンのアルゴリズム - Ford – Fulkerson algorithm | 217 |
| 20.2 | 素なパス - Disjoint paths | 220 |
| 20.3 | 最大マッチング - Maximum matchings | 222 |
| 20.4 | 辺被覆 - Path covers | 225 |
| 第 III 部 | 発展的なテーマ - Advanced topics | 229 |
| 第 21 章 | 整数論 - Number theory | 231 |
| 21.1 | 素数と因数 - Primes and factors | 231 |
| 21.2 | mod の計算 - Modular arithmetic | 236 |
| 21.3 | 方程式を解く - Solving equations | 239 |
| 21.4 | その他 - Other results | 241 |
| 第 22 章 | 組合わせ論 - Combinatorics | 243 |
| 22.1 | 二項係数 - Binomial coefficients | 244 |
| 22.2 | カタラン数 - Catalan numbers | 247 |
| 22.3 | 包除原理 - Inclusion-exclusion | 249 |

| | | |
|--------|---|-----|
| 22.4 | Burnside's lemma | 251 |
| 22.5 | Cayley's formula | 252 |
| 第 23 章 | 行列 - Matrices | 255 |
| 23.1 | 行列の演算 - Operations | 256 |
| 23.2 | 線型回帰 - Linear recurrences | 258 |
| 23.3 | グラフと行列 - Graphs and matrices | 260 |
| 第 24 章 | 確率 - Probability | 265 |
| 24.1 | 確率の計算 - Calculation | 265 |
| 24.2 | 事象 - Events | 266 |
| 24.3 | 確率変数 - Random variables | 269 |
| 24.4 | マルコフ連鎖 - Markov chains | 271 |
| 24.5 | 乱択 | 272 |
| 第 25 章 | ゲーム理論 | 275 |
| 25.1 | ゲームの状態 - Game states | 275 |
| 25.2 | Nim - Nim game | 277 |
| 25.3 | Sprague - Grundy theorem | 279 |
| 第 26 章 | 文字列アルゴリズム - String algorithms | 285 |
| 26.1 | 文字列に関する用語 - String terminology | 285 |
| 26.2 | トライ構造 - Trie structure | 286 |
| 26.3 | 文字列のハッシュ化 - String hashing | 288 |
| 26.4 | Z-アルゴリズム - Z-algorithm | 291 |
| 第 27 章 | 平方根アルゴリズム - Square root algorithms | 295 |
| 27.1 | アルゴリズムの組み合わせ - Combining algorithms | 296 |
| 27.2 | 整数のパーティション - Integer partitions | 298 |
| 27.3 | Mo のアルゴリズム - Mo's algorithm | 300 |
| 第 28 章 | Segment trees revisited | 303 |
| 28.1 | Lazy propagation | 304 |
| 28.2 | Dynamic trees | 308 |
| 28.3 | Data structures | 310 |
| 28.4 | Two-dimensionality | 311 |
| 第 29 章 | Geometry | 313 |

| | | |
|--------------|---|-----|
| 29.1 | Complex numbers | 314 |
| 29.2 | Points and lines | 316 |
| 29.3 | Polygon area | 320 |
| 29.4 | Distance functions | 321 |
| 第 30 章 | 掃引線アルゴリズム - Sweep line algorithms | 325 |
| 30.1 | 交差点 - Intersection points | 326 |
| 30.2 | 近接ペア問題 - Closest pair problem | 327 |
| 30.3 | 凸包問題 - Convex hull problem | 329 |
| 参考文献 | | 331 |
| Bibliography | | 331 |

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地震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

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次の二つのトピックからなっている。アルゴリズムの設計とアルゴリズムの実装である。

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、想像的に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、多くの問題でポイントとなるのは効率的なアルゴリズムの考察です。

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からなります。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研究の基礎でもあります。

The アルゴリズムの実装 にはプログラミングスキルが大切になります。競技プログラミングでは、実装したアルゴリズムはテストケースでテストすることで採点されます。このため、アルゴリズムの考察に加えて、その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 人をみると、79 % が C++, 16 % が Python, 8 % が Java を使用しています [29]。また、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++ はほぼどのコンテストでも利用できます。C++ を使う利点は次の通りです。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。

一方で複数の言語を使いこなし、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、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。とはいえ、コンテストの多くではある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。

All example programs in this book are written in C++, and the standard library's data structures and algorithms are often used. The programs follow the C++11 standard, which can be used in most contests nowadays. If you cannot program in C++ yet, now is a good time to start learning. 本書で紹介するプログラムは C++11 で書かれており、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 (訳註：2018 年は C++17 対応のコンテストサイトは少なかった)。標準ライブラリのデータ構造やアルゴリズムが多く使用されています。C++ でプログラミングができない人は、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。つまり、よく使う iostream, vector や algorithm, などが使えるようになります。

using 行は標準ライブラリの機能を一括で使えるようにします。using がない場合は std::cout と書かないと行けませんが、これがあることで cout だけで十分になります。

そして、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 を出すことを示します。

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。さら
 に C の関数である scanf と printf も利用できます。

通常、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おり、これ
 らは以下のように cin で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
```

The 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);
```

"\n"用いると endl よりも高速です。なぜなら、endl は毎回 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
 の関数は少し速いに動作しますが、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);
```

また、次の通り出力します。

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

文字列の読み込みでは空白ごと読み込みたいことがあり、これは `getline` 関数を使います。

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

また、読み込みたい文字列の数がわからない場合は次のように処理します。

```
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。

In some contest systems, files are used for input and output. An easy solution for this is to write the code as usual using standard streams, but add the following lines to the beginning of the code:

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

After this, the program reads the input from the file "input.txt" and writes the output to the file "output.txt".

1.3 数字を扱う - Working with numbers

整数 - Integers

競技プログラミングで最もよく使われる整数型は `int` という 32 ビットの型です。値の範囲は $-2^{31} \dots 2^{31}-1$ 、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、 $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$ です。

次のコードでは、long long 変数を定義しています。

long long variable:

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

数字の後の LL はこの数字が long long であるということです。

long long を使う時によくある過ちは int との演算をしてしまうケースです。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 ですが、a*a は int 同士の掛け算で、この結果は int となります。そして、(int で表現できる範囲を超えてしまうので) 誤った結果となります。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ことで正しく動作します。

通常、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おくと、値域が $-2^{127} \dots 2^{127} - 1$ 、つまり、 $-10^{38} \dots 10^{38}$ の範囲を扱えます。しかし、この型は、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です。例えば、 $17 \bmod 5 = 2$ です。なぜなら、 $17 = 3 \cdot 5 + 2$ であるため。

答えの数が非常に大きくなる問題では、時々「 m で割った数を答えよ」と指示されます。例えば、 $10^9 + 7$ などがあります。これにより答えが非常に大きくても int や long long の値域で十分になります。

余りの重要な性質は、足し算、引き算、掛け算において、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 $\text{mod } m$ で求めます。

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

通常、余りは $0 \dots m-1$ で表現されます。しかし、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います。

これを解決するにはまずあまりを求め、マイナスであった場合は m を加算することです。

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

これはあくまで負の可能性となることがある場合にのみ必要になります。

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、g++ の拡張である 80-bit 浮動小数点型の `long double` もあります。ほとんどのケースでは `double` で十分ですが、`long double` の方がより正確です。

答えに必要な精度は、通常、問題文の中で示されています。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す。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っ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驚くべきものでしょう。

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

正しい値は 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います。このまま `==` 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 ($\epsilon = 10^{-9}$)。

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

なお、ある程度までの整数は正確に表現できます。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、速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、競技志向のプログラマは、データ型などのコードに短い名前を定義することが多くなります。

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";
```

となっていたコードが以下のようにかけます。

```
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かけます。

```
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。

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

このような平方根を求めるものがあり、

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

こうしたすると、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)
```

この時、

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

この処理は次のように動作し、予想した通りに動きますね。

```
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できません。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っている。例えば^{*1},

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

や

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

^{*1} There is even a general formula for such sums, called **Faulhaber's formula**, but it is too complex to be presented here.

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。

$$3, 7, 11, 15$$

この場合、差は常に 4 です。等差数列の和は次の式で表されます。

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

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。

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

この式は、和が n 個の数からなり、各数の値が平均して $(a + b)/2$ であることから算出されます。

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。

$$3, 6, 12, 24$$

というのは常に 2 倍になっています。この話は次のように求められます。

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

a は初項、 b は最後の項、 k は項の比です。

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

これは次のように求められます。

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

両辺に k をかけます。

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

これを整理して

$$kS - S = bk - a$$

を餅挽きます。

また、特殊なケースで以下の公式が成り立ちます。

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

調和級数 - **harmonic sum** は次のような式のことです。

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

An upper bound for a harmonic sum is $\log_2(n) + 1$. Namely, we can modify each term $1/k$ so that k becomes the nearest power of two that does not exceed k . For example, when $n = 6$, we can estimate the sum as follows:

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

This upper bound consists of $\log_2(n) + 1$ parts ($1, 2 \cdot 1/2, 4 \cdot 1/4$, etc.), and the value of each part is at most 1.

集合論 - Set theory

set とは要素のコレクションのことです。例えば set である

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

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、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。

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

集合へ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、 $A \cap B = \{2\}$.
- **The 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、 $A \cup B = \{2, 3, 7, 8\}$.
- **The 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 $\bar{A} = \{3, 4, 6, 8, 9, 10\}$ となります。
- **The 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 の各要素が S にも属する場合、 A は S の**部分集合 - subset** であると呼べて、

$A \subset S$ 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。

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

よく、次のような集合が使われます。 \mathbb{N} (自然数 - natural numbers), \mathbb{Z} (整数 - integers), \mathbb{Q} (有理数 - rational numbers) and \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。

次のような集合を定義しても良いです。

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

$f(n)$ はなんらかの関数です。これは、 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 **true** (1) か **false** (0) で表されます (訳注: 日本語では真と偽ともいいます)。代表的な論理式の記号を示します。 \neg (**negation**), \wedge (**conjunction**), \vee (**disjunction**), \Rightarrow (**implication**), \Leftrightarrow (**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 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。

A **predicate** is an expression that is true or false depending on its parameters. Predicates are usually denoted by capital letters. For example, we can define a predicate $P(x)$ that is true exactly when x is a prime number. Using this definition, $P(7)$ is true but $P(8)$ is false.

A **quantifier** connects a logical expression to the elements of a set. The most important quantifiers are \forall (**for all**) and \exists (**there is**). For example,

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

means that for each element x in the set, there is an element y in the set such that y is smaller than x . This is true in the set of integers, but false in the set of natural numbers.

Using the notation described above, we can express many kinds of logical propositions. For example,

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

means that if a number x is larger than 1 and not a prime number, then there are numbers a and b that are larger than 1 and whose product is x . This proposition is true in the set of integers.

Functions

The function $\lfloor x \rfloor$ rounds the number x down to an integer, and the function $\lceil x \rceil$ rounds the number x up to an integer. For example,

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

The functions $\min(x_1, x_2, \dots, x_n)$ and $\max(x_1, x_2, \dots, x_n)$ give the smallest and largest of values x_1, x_2, \dots, x_n . For example,

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

The **factorial** $n!$ can be defined

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

or recursively

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

The **Fibonacci numbers** arise in many situations. They can be defined recursively as follows:

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

The first Fibonacci numbers are

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

There is also a closed-form formula for calculating Fibonacci numbers, which is sometimes called **Binet's formula**:

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

Logarithms

The **logarithm** of a number x is denoted $\log_k(x)$, where k is the base of the logarithm. According to the definition, $\log_k(x) = a$ exactly when $k^a = x$.

A useful property of logarithms is that $\log_k(x)$ equals the number of times we have to divide x by k before we reach the number 1. For example, $\log_2(32) = 5$ because 5 divisions by 2 are needed:

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

Logarithms are often used in the analysis of algorithms, because many efficient algorithms halve something at each step. Hence, we can estimate the efficiency of such algorithms using logarithms.

The logarithm of a product is

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

and consequently,

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

In addition, the logarithm of a quotient is

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

Another useful formula is

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

and using this, it is possible to calculate logarithms to any base if there is a way to calculate logarithms to some fixed base.

The **natural logarithm** $\ln(x)$ of a number x is a logarithm whose base is $e \approx 2.71828$. Another property of logarithms is that the number of digits of an integer x in base b is $\lfloor \log_b(x) \rfloor + 1$. For example, the representation of 123 in base 2 is 1111011 and $\lfloor \log_2(123) \rfloor + 1 = 7$.

1.6 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

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

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]
- 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。問題をゆっくり解く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。

さて、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ます。時間計算量によってアルゴリズムが十分に速いかどうかを実装せずに知ることができます。

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、 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、 $\lceil n/2 \rceil$ 回実行されますが、それぞれのコードの時間複雑度は $O(n)$ です。

A time complexity does not tell us the exact number of times the code inside a loop is executed, but it only shows the order of magnitude. In the following examples, the code inside the loop is executed $3n$, $n+5$ and $\lceil n/2 \rceil$ times, but the time complexity of each code is $O(n)$.

```
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。これは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、時間的な複雑さを持つ 3 つのフェーズから構成されていて、 $O(n)$, $O(n^2)$, $O(n)$ からなります。この場合、この時間計算量は $O(n^2)$ となります。

```
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、いくつか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なるでしょう。

例えば、次のような関数を考えます。

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

```
f(n-1);
}
```

$f(n)$ 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} |

このように、この場合は以下のような時間計算量になります。

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

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 になる回数に等しいためです。

平方根アルゴリズム - $O(\sqrt{n})$ 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: なんか変)

$O(n)$ **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、これは可能な限り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はこの計算量になります。あるいは、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。

$O(n^2)$ **二次 - 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。

$O(n^3)$ **三次 - 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。

$O(2^n)$ これは入力に対して全ての集合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 は通常小さいので、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。ですが、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-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。留意すべきは現代のコンピュータが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

*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].

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。

一方、入力サイズがあれば、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。

| 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 $O(\log n)$ |

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。この情報は、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ことは忘れないでください。例えば、 $O(n)$ 時間で実行されるアルゴリズムは、 $n/2$ 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ます。しかし、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興味深い問題となります。

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

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

この場合の最大の部分和は 10 です。

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

なお、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ります。

Algorithm 1

この問題を解決する簡単な方法は、可能性のあるすべての部分配列を調べ、それぞれの部分配列の値の合計を計算し、最大合計を計算することです。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 はこの最大値を持つものです。

これは、ネストから推測できる通り、 $O(n^3)$ の計算量となります。

Algorithm 2

簡単に先ほどのコードからループを 1 つ撮ることができます。右端を動かす時に和も計算して終えればいいのです。

```
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。

Algorithm 3

驚くべきことに、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。この後、問題の答えは、それらの最大値となります。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

後者の場合、総和が最大となる部分配列を求めたいので、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。

実装を示します。

```
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";
```

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's algorithm**.

| array size n | Algorithm 1 | Algorithm 2 | Algorithm 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効率的だが、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。アルゴリズム 1 は $n = 10^4$ のときに遅くなり、アルゴリズム 2 は $n = 10^5$ のときに遅くなりました。アルゴリズム 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 - Sorting

ソート - 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。なぜなら、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。もし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で、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、それらは様々なアルゴリズムを適用する良い例にもなっています。効率的な一般的な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。

For example, the array

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

これを次のように操作します。

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

$O(n^2)$ algorithms

配列をソートするための簡単なアルゴリズムは、 $O(n^2)$ 時間で動作します。このような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、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つの要素のうち、順序が正しくないものが見つかったらそれを交換します。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、最大の値が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。

ところで、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). 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。もし、転置がないとき、配列は完全にソートされています。一方、配列の要素が逆順の場合、逆順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。したがって、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、各入れ替えは最大でも 1 つの反転を取り除き、アルゴリズムは少なくとも $O(n^2)$ です。

$O(n \log n)$ algorithms

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. If $a = b$, 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. 再起的に $\text{array}[a \dots k]$ をソートする
4. 再起的に $\text{array}[k + 1 \dots b]$ 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

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して、効率的に動作す

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

るアルゴリズムです。再帰は $O(\log n)$ の階層で行われ、それぞれの階層では $O(n)$ 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。as follows:

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

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

そして、それぞれは次のようにソートされます。

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

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

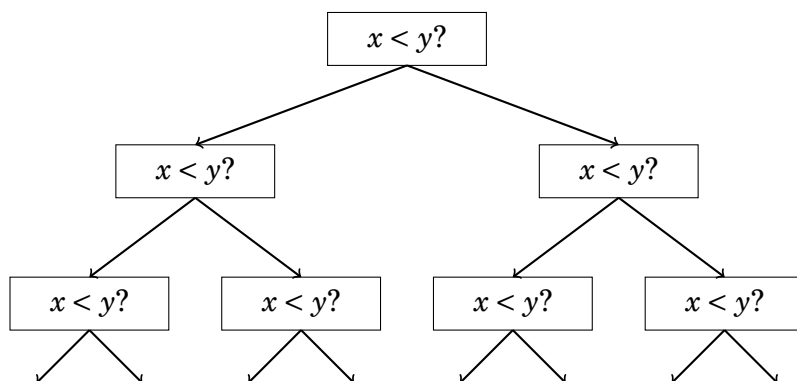
最後にその2つのソートされた配列を結合します。

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

ソート時間の限界 - Sorting lower bound

さて、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、それは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すことで、時間計算量の限界を証明することができます。この処理により、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。このため、木の高

さは少なくとも次の通りです。

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

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。この結果、推定ができ、

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

このように、 $n \log n$ がソートの限界であることがわかります。

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、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 |
|---|---|---|---|---|---|---|---|

この配列があった時に、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 |

例えば、このカウントの3の位置の値は2であり、これは要素3が元の配列に2回現れる、ということを意味します。この構築には $O(n)$ の時間がかかり、各要素の出現回数をブックキーピング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。例えば、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。第二に、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());
```

この結果配列自身は次のようになります。[2,3,3,4,5,5,8]. 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());
```

なお、`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、2つの要素の順序を調べる必要がある場合は常にこの演算子が使用されます。

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) に従ってソートされます。

```
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());
```

この結果、`vector` は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<の演算子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）。

例えば、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ので、この構造体はまず x で比較され、同点なら y で判定を行います。

```
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、おなじ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;
}
```

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、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、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、この方法も最良の方法と言えます。

ところが配列がソートされている状況では良い方法があり、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。

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。

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、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。そして、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。例えば、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
}
```

また、これらを活用することで次のコードは 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

二分探索の重要な動作は関数の値が変化する位置を見つけることである。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。与えられている。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。

ここで、次のように示せます。

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

今、 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。したがって、次の値 $k = x+1$ は、 $ok(x)$ が真となる最小の値といえます。最初のジャンプの長さ z は十分大きくなければなりません。 $ok(z)$ 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 に依存する。例えば、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、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。今、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。

これは $k = x + 1$ 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ダメなことに注意してください。そのような場合はどちらに探索を続ければいいのかがわからなくなってしまいます。

第 4 章

データ構造 - Data structures

A **データ構造 -data structure** とは、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、問題に適したデータ構造を選択することが大切になります。重要なのは、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。可能な限り標準ライブラリを使用することは、多くの時間を節約することができるのでベストな方法と言えます。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、ほとんど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]
```

この結果、以下のように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。次のように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
```

また次のように配列を要素をもった状態で作成することもできます。

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

また、**vector** は次のようにして要素数を指定して作成、また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、新しい配列を確保しすべての要素を新しい配列に移動させます。ただし、これらがあまり起こることはないので push_back の平均的な時間複雑度は $O(1)$ と考えてよいです。

また、string 構造体は動的な配列で、ほとんどベクトル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構文があります。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、ある部分文字列 t が最初に出現する位置を探します。

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 つの集合の実装が含まれている。1 つ目は `set` で、平衡二分木に基づいており、その演算は $O(\log n)$ 時間で行われます。もう一つは `unordered_set` でハッシュを使用し、その演算は平均して $O(1)$ 時間で動作する。

どのセット実装を使うかは、しばしば好みの問題である。`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できます。一方、`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。

```
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
```

セットはほとんどベクトルのように使うことができますが、[] 記法を使って要素に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、すべての要素を反復処理するものです。

```
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 *distinct* であることです。したがって、関数 `count` は常に 0（要素が集合にない）か 1（要素が集合にある）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。次のコードでこれを説明しましょう。

```
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` と同様に機能しますが、ある要素について複数のインスタンス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` は複数の要素があったとしても全てを削除します。

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

この時、1 つの要素だけを削除死体のなら次のようにします。

```
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でよく、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 $O(\log n)$ 時間かかる。`unordered_map` はハッシュを使用しており、要素へのアクセスに平均で $O(1)$ 時間かかる。

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["harpsichord"] = 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) 例えば、次のコードでは、値 0 を持つキー "aybaltu" がマップに追加されます。

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

`count` はキーの存在を確認します。

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

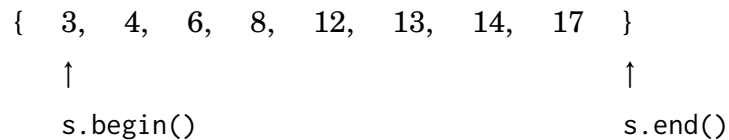
次のコードはキーと値を列挙します。

```
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。

よく使われるイテレータ `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。次の図がイメージです。



`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* です。

範囲の利用 - Working with ranges

イテレータは、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` が関数に与えられます。

例えば、次のコードは、`sort` 関数でベクトルをソートし、`reverse` 関数で要素の順序を反転させ、最後に `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

Bitset

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が1ビットのメモリしか使用しないため、通常の配列よりも少ないメモリしか必要としないことです。例えば、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、対応するビットセットは 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、上記の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出来ます。

```
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。

例を示します。

```
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、優先キューのすべての操作が可能ですが、優先度付きキューを使用する利点は定数係数がより小さいことです。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、これは、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。

次のコードでこれを説明します。

```
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創りたいならば

```
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;
```

この後、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ます。int に対する定義は以下の通りである。

```
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 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 \log 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。時には、これらのアプローチの実際の効率に顕著な違いがあり、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死体とします。

例えば、以下の例を考えます。

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

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いきます。

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するのである。このアルゴリズムの時間計算量は $O(n \log n)$ です。

Algorithm 2

順序付きセット出なくともよいので、set の代わりに unordered_set を使うこともできます。これは、基礎となるデータ構造を変更するだけなので、アルゴリズムをより効率的にする簡単な方法です。このアルゴリズムの時間計算量は $O(n)$ である。

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 | Algorithm 1 | Algorithm 2 | Algorithm 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、最も効率的なアルゴリズムは、ソートを使用するアルゴリズム 3 です。アルゴリズム 2 に比べ、半分の時間しか使いません。興味深いことに、アルゴリズム 1 とアルゴリズム 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アルゴリズム 3 は 10 倍も速いのです。

これは、ソートがアルゴリズム 3 の冒頭で一度だけ行われるのに対して、残りのアルゴリズムは線形時間で動作からです。

一方、Algorithm 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 $O(\log n)$ の動作を行うからです。

第 5 章

全探索 - Complete search

全探索 - Complete search は、ほとんどすべてのアルゴリズムの問題を解決するために使用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。完全探索は、すべての解を調べるのに十分な時間があればとても手法です。

なぜなら、この探索は通常、他の解法に比べて実装が簡単で、常に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、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ベクトル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。

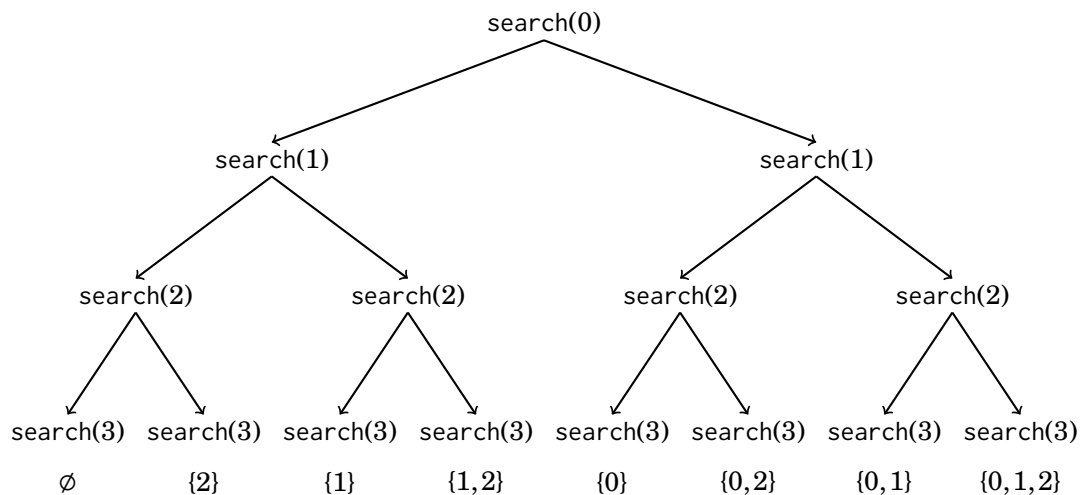
```
void search(int k) {  
    if (k == n) {  
        // process subset  
    } else {
```

```

        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。そして、 $k=n$ 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終わります。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 は、どの要素が部分集合に含まれるかを示す。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 ですが、これは部分集合 $\{0,3,4\}$ 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ベクトルを返します。

```
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のどちらかである。

Method 1

並べ換えも再帰を使って生成できます。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、並べ換えを含むベクターを構築し、パラメータなしでこの関数が呼ばれたときに探索が開始されます。

```
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、新しいエレメントが `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格納します。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

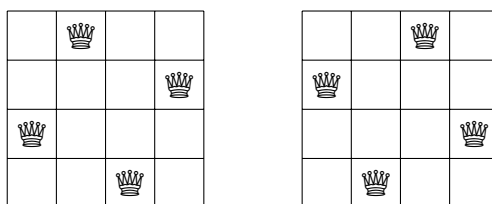
もう一つの方法は、この順列を $\{0, 1, \dots, n-1\}$ から開始することです。そして、次の順列、を求めていくことです。C++ の標準ライブラリにはこのための関数 `next_permutation` があり、次のように使うことができます。

```
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

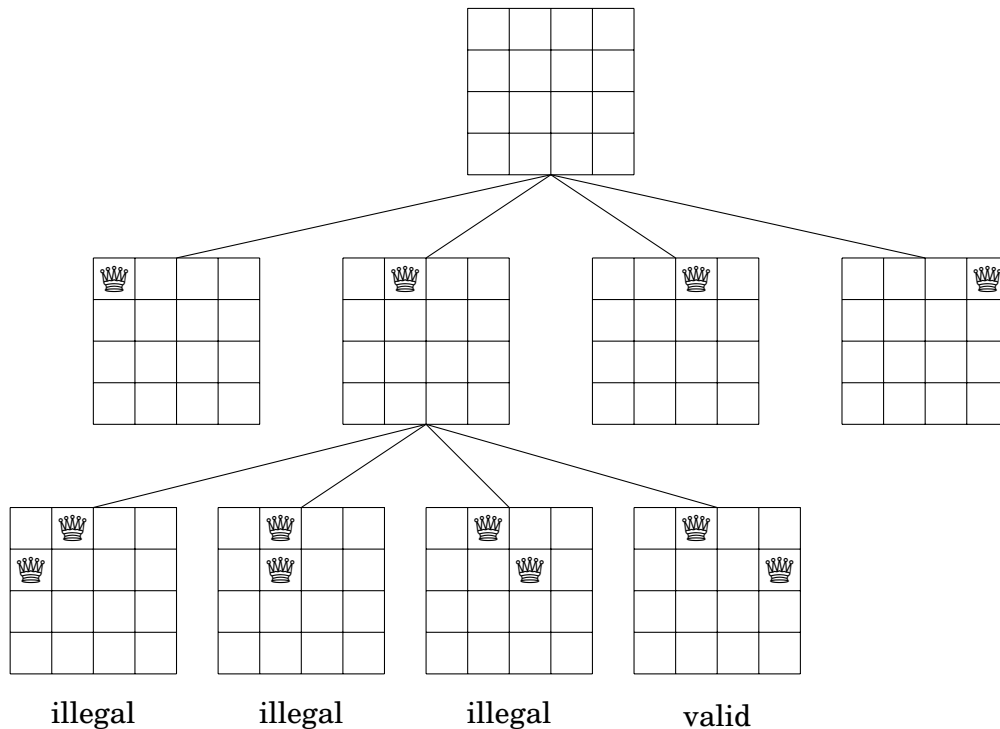
バックトラッキング - backtracking のアルゴリズムとは、空の解から始まり、段階的に解を作っていきます。探索は、解がどのように構築されうるかを再帰で全探索していきます。

例えば、 $n \times n$ 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、 n 個の女王を配置する方法の数を計算する問題を考えます。これは、クイーン問題として知られています。例えば、 $n = 4$ のとき、2つの解が考えられる。



クイーン問題は、バックトラックを使用して、ボードに一行ずつクイーンを配置することで解くことができます。各ターンでは、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。



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、不正な構成です。しかし、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)`. を呼び出ししてこの探索は開始されます。盤の大きさは $n \times n$ であ、コードは数えるべき解の数を計算します。

さて、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 行にクイーンを置き、内

部ではパラメータ $y+1$ 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 が 1 増加します。

配列 column はクイーンを含む列を、配列 diag1 および diag2 は対角線を追跡します。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されています。

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |

column

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

diag1

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

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 $q(n)$ としましょう。

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、解の数が指数関数的に増えるので、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、最近のコンピュータでは既に約 1 分かかっている¹。

Let $q(n)$ denote the number of ways to place n queens on an $n \times n$ chessboard. The above backtracking algorithm tells us that, for example, $q(8) = 92$. When n increases, the search quickly becomes slow, because the number of solutions increases exponentially. For example, calculating $q(16) = 14772512$ using the above algorithm already takes about a minute on a modern computer^{*1}.

5.4 Pruning the search

We can often optimize backtracking by pruning the search tree. The idea is to add "intelligence" to the algorithm so that it will notice as soon as possible if a partial solution cannot be extended to a complete solution. Such optimizations can have a tremendous effect on the efficiency of the search.

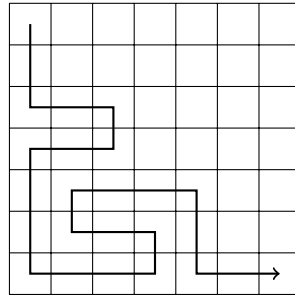
Let us consider the problem of calculating the number of paths in an $n \times n$ grid from the upper-left corner to the lower-right corner such that the path visits each square exactly once. For example, in a 7×7 grid, there are 111712 such paths. One of the paths is as follows:

^{*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].

- number of recursive calls: 38 billion

Optimization 2

If the path reaches the lower-right square before it has visited all other squares of the grid, it is clear that it will not be possible to complete the solution. An example of this is the following path:

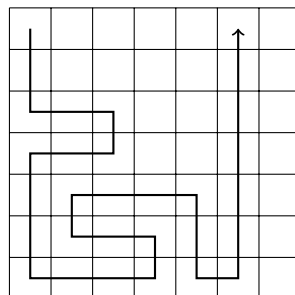


Using this observation, we can terminate the search immediately if we reach the lower-right square too early.

- running time: 119 seconds
- number of recursive calls: 20 billion

Optimization 3

If the path touches a wall and can turn either left or right, the grid splits into two parts that contain unvisited squares. For example, in the following situation, the path can turn either left or right:

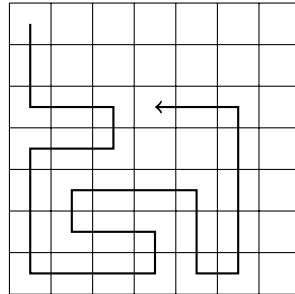


In this case, we cannot visit all squares anymore, so we can terminate the search. This optimization is very useful:

- running time: 1.8 seconds
- number of recursive calls: 221 million

Optimization 4

The idea of Optimization 3 can be generalized: if the path cannot continue forward but can turn either left or right, the grid splits into two parts that both contain unvisited squares. For example, consider the following path:



It is clear that we cannot visit all squares anymore, so we can terminate the search. After this optimization, the search is very efficient:

- running time: 0.6 seconds
- number of recursive calls: 69 million

Now is a good moment to stop optimizing the algorithm and see what we have achieved. The running time of the original algorithm was 483 seconds, and now after the optimizations, the running time is only 0.6 seconds. Thus, the algorithm became nearly 1000 times faster after the optimizations.

This is a usual phenomenon in backtracking, because the search tree is usually large and even simple observations can effectively prune the search. Especially useful are optimizations that occur during the first steps of the algorithm, i.e., at the top of the search tree.

5.5 Meet in the middle

Meet in the middle is a technique where the search space is divided into two parts of about equal size. A separate search is performed for both of the parts, and finally the results of the searches are combined.

The technique can be used if there is an efficient way to combine the results of the searches. In such a situation, the two searches may require less time than one large search. Typically, we can turn a factor of 2^n into a factor of $2^{n/2}$ using the meet in the middle technique.

As an example, consider a problem where we are given a list of n numbers and a number x , and we want to find out if it is possible to choose some numbers from the list so that their sum is x . For example, given the list $[2, 4, 5, 9]$ and $x = 15$, we can choose the numbers $[2, 4, 9]$ to get $2 + 4 + 9 = 15$. However, if $x = 10$ for the same list, it is not possible to form the sum.

A simple algorithm to the problem is to go through all subsets of the elements and check if the sum of any of the subsets is x . The running time of such an algorithm is $O(2^n)$, because there are 2^n subsets. However, using the meet in the middle technique, we can achieve a more efficient $O(2^{n/2})$ time algorithm^{*2}. Note that $O(2^n)$ and $O(2^{n/2})$ are different complexities because $2^{n/2}$ equals $\sqrt{2^n}$.

The idea is to divide the list into two lists A and B such that both lists contain about half of the numbers. The first search generates all subsets of A and stores their sums to a list S_A . Correspondingly, the second search creates a list S_B from B . After this, it suffices to check if it is possible to choose one element from S_A and another element from S_B such that their sum is x . This is possible exactly when there is a way to form the sum x using the numbers of the original list.

For example, suppose that the list is $[2, 4, 5, 9]$ and $x = 15$. First, we divide the list into $A = [2, 4]$ and $B = [5, 9]$. After this, we create lists $S_A = [0, 2, 4, 6]$ and $S_B = [0, 5, 9, 14]$. In this case, the sum $x = 15$ is possible to form, because S_A contains the sum 6, S_B contains the sum 9, and $6 + 9 = 15$. This corresponds to the solution $[2, 4, 9]$.

We can implement the algorithm so that its time complexity is $O(2^{n/2})$. First, we generate *sorted* lists S_A and S_B , which can be done in $O(2^{n/2})$ time using a merge-like technique. After this, since the lists are sorted, we can check in $O(2^{n/2})$ time if the sum x can be created from S_A and S_B .

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

第 6 章

Greedy algorithms

A **greedy algorithm** constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.

The difficulty in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It is often difficult to argue that a greedy algorithm works.

6.1 Coin problem

As a first example, we consider a problem where we are given a set of coins and our task is to form a sum of money n using the coins. The values of the coins are $\text{coins} = \{c_1, c_2, \dots, c_k\}$, and each coin can be used as many times we want. What is the minimum number of coins needed?

For example, if the coins are the euro coins (in cents)

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

and $n = 520$, we need at least four coins. The optimal solution is to select coins $200 + 200 + 100 + 20$ whose sum is 520.

Greedy algorithm

A simple greedy algorithm to the problem always selects the largest possible coin, until the required sum of money has been constructed. This algorithm works in the example case, because we first select two 200 cent coins, then one

100 cent coin and finally one 20 cent coin. But does this algorithm always work?

It turns out that if the coins are the euro coins, the greedy algorithm *always* works, i.e., it always produces a solution with the fewest possible number of coins. The correctness of the algorithm can be shown as follows:

First, each coin 1, 5, 10, 50 and 100 appears at most once in an optimal solution, because if the solution would contain two such coins, we could replace them by one coin and obtain a better solution. For example, if the solution would contain coins $5 + 5$, we could replace them by coin 10.

In the same way, coins 2 and 20 appear at most twice in an optimal solution, because we could replace coins $2 + 2 + 2$ by coins $5 + 1$ and coins $20 + 20 + 20$ by coins $50 + 10$. Moreover, an optimal solution cannot contain coins $2 + 2 + 1$ or $20 + 20 + 10$, because we could replace them by coins 5 and 50.

Using these observations, we can show for each coin x that it is not possible to optimally construct a sum x or any larger sum by only using coins that are smaller than x . For example, if $x = 100$, the largest optimal sum using the smaller coins is $50 + 20 + 20 + 5 + 2 + 2 = 99$. Thus, the greedy algorithm that always selects the largest coin produces the optimal solution.

This example shows that it can be difficult to argue that a greedy algorithm works, even if the algorithm itself is simple.

General case

In the general case, the coin set can contain any coins and the greedy algorithm *does not* necessarily produce an optimal solution.

We can prove that a greedy algorithm does not work by showing a counterexample where the algorithm gives a wrong answer. In this problem we can easily find a counterexample: if the coins are $\{1, 3, 4\}$ and the target sum is 6, the greedy algorithm produces the solution $4 + 1 + 1$ while the optimal solution is $3 + 3$.

It is not known if the general coin problem can be solved using any greedy algorithm^{*1}. However, as we will see in Chapter 7, in some cases, the general problem can be efficiently solved using a dynamic programming algorithm that always gives the correct answer.

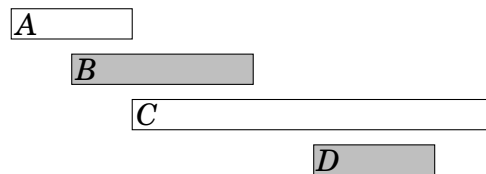
^{*1} However, it is possible to *check* in polynomial time if the greedy algorithm presented in this chapter works for a given set of coins [53].

6.2 Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following events:

| event | starting time | ending time |
|----------|---------------|-------------|
| <i>A</i> | 1 | 3 |
| <i>B</i> | 2 | 5 |
| <i>C</i> | 3 | 9 |
| <i>D</i> | 6 | 8 |

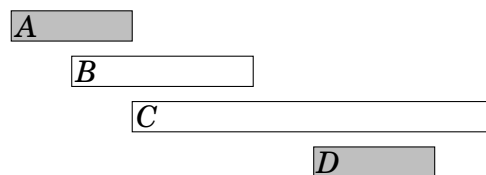
In this case the maximum number of events is two. For example, we can select events *B* and *D* as follows:



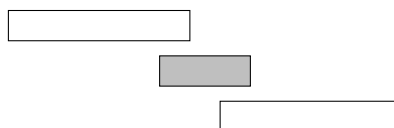
It is possible to invent several greedy algorithms for the problem, but which of them works in every case?

Algorithm 1

The first idea is to select as *short* events as possible. In the example case this algorithm selects the following events:



However, selecting short events is not always a correct strategy. For example, the algorithm fails in the following case:

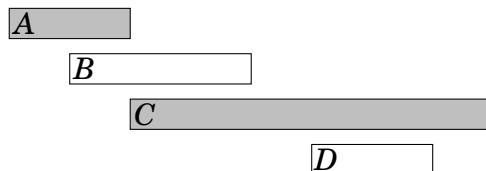


If we select the short event, we can only select one event. However, it would be

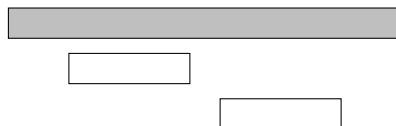
possible to select both long events.

Algorithm 2

Another idea is to always select the next possible event that *begins* as *early* as possible. This algorithm selects the following events:



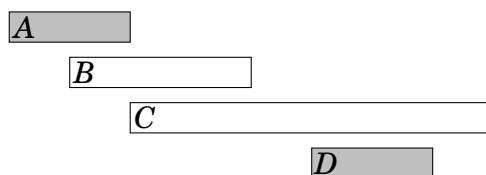
However, we can find a counterexample also for this algorithm. For example, in the following case, the algorithm only selects one event:



If we select the first event, it is not possible to select any other events. However, it would be possible to select the other two events.

Algorithm 3

The third idea is to always select the next possible event that *ends* as *early* as possible. This algorithm selects the following events:



It turns out that this algorithm *always* produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events.

One way to argue that the algorithm works is to consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

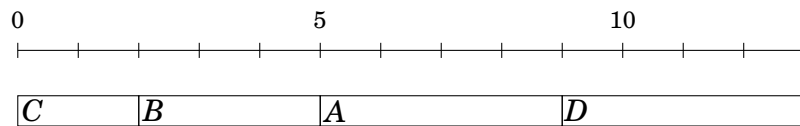
6.3 Tasks and deadlines

Let us now consider a problem where we are given n tasks with durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn $d - x$ points where d is the task's deadline and x is the moment when we finish the task. What is the largest possible total score we can obtain?

For example, suppose that the tasks are as follows:

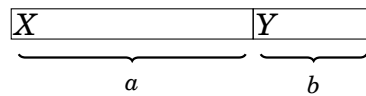
| task | duration | deadline |
|------|----------|----------|
| A | 4 | 2 |
| B | 3 | 5 |
| C | 2 | 7 |
| D | 4 | 5 |

In this case, an optimal schedule for the tasks is as follows:

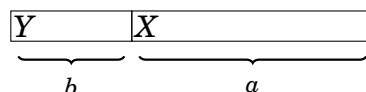


In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10 .

Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks *sorted by their durations* in increasing order. The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks. For example, consider the following schedule:



Here $a > b$, so we should swap the tasks:



Now X gives b points less and Y gives a points more, so the total score increases by $a - b > 0$. In an optimal solution, for any two consecutive tasks, it must hold that the shorter task comes before the longer task. Thus, the tasks must be

performed sorted by their durations.

6.4 Minimizing sums

We next consider a problem where we are given n numbers a_1, a_2, \dots, a_n and our task is to find a value x that minimizes the sum

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

We focus on the cases $c = 1$ and $c = 2$.

Case $c = 1$

In this case, we should minimize the sum

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

For example, if the numbers are $[1, 2, 9, 2, 6]$, the best solution is to select $x = 2$ which produces the sum

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

In the general case, the best choice for x is the *median* of the numbers, i.e., the middle number after sorting. For example, the list $[1, 2, 9, 2, 6]$ becomes $[1, 2, 2, 6, 9]$ after sorting, so the median is 2.

The median is an optimal choice, because if x is smaller than the median, the sum becomes smaller by increasing x , and if x is larger than the median, the sum becomes smaller by decreasing x . Hence, the optimal solution is that x is the median. If n is even and there are two medians, both medians and all values between them are optimal choices.

Case $c = 2$

In this case, we should minimize the sum

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

For example, if the numbers are $[1, 2, 9, 2, 6]$, the best solution is to select $x = 4$ which produces the sum

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

In the general case, the best choice for x is the *average* of the numbers. In the example the average is $(1 + 2 + 9 + 2 + 6)/5 = 4$. This result can be derived by presenting the sum as follows:

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

The last part does not depend on x , so we can ignore it. The remaining parts form a function $nx^2 - 2xs$ where $s = a_1 + a_2 + \cdots + a_n$. This is a parabola opening upwards with roots $x = 0$ and $x = 2s/n$, and the minimum value is the average of the roots $x = s/n$, i.e., the average of the numbers a_1, a_2, \dots, a_n .

6.5 Data compression

A **binary code** assigns for each character of a string a **codeword** that consists of bits. We can *compress* the string using the binary code by replacing each character by the corresponding codeword. For example, the following binary code assigns codewords for characters A – D:

| character | codeword |
|-----------|----------|
| A | 00 |
| B | 01 |
| C | 10 |
| D | 11 |

This is a **constant-length** code which means that the length of each codeword is the same. For example, we can compress the string AABACDACA as follows:

000001001011001000

Using this code, the length of the compressed string is 18 bits. However, we can compress the string better if we use a **variable-length** code where codewords may have different lengths. Then we can give short codewords for characters that appear often and long codewords for characters that appear rarely. It turns out that an **optimal** code for the above string is as follows:

| character | codeword |
|-----------|----------|
| A | 0 |
| B | 110 |
| C | 10 |
| D | 111 |

An optimal code produces a compressed string that is as short as possible. In this case, the compressed string using the optimal code is

001100101110100,

so only 15 bits are needed instead of 18 bits. Thus, thanks to a better code it was possible to save 3 bits in the compressed string.

We require that no codeword is a prefix of another codeword. For example, it is not allowed that a code would contain both codewords 10 and 1011. The reason for this is that we want to be able to generate the original string from the compressed string. If a codeword could be a prefix of another codeword, this would not always be possible. For example, the following code is *not* valid:

| character | codeword |
|-----------|----------|
| A | 10 |
| B | 11 |
| C | 1011 |
| D | 111 |

Using this code, it would not be possible to know if the compressed string 1011 corresponds to the string AB or the string C.

Huffman coding

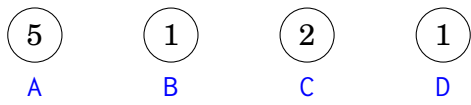
Huffman coding^{*2} is a greedy algorithm that constructs an optimal code for compressing a given string. The algorithm builds a binary tree based on the frequencies of the characters in the string, and each character's codeword can be read by following a path from the root to the corresponding node. A move to the left corresponds to bit 0, and a move to the right corresponds to bit 1.

Initially, each character of the string is represented by a node whose weight is the number of times the character occurs in the string. Then at each step two nodes with minimum weights are combined by creating a new node whose weight is the sum of the weights of the original nodes. The process continues until all nodes have been combined.

Next we will see how Huffman coding creates the optimal code for the string AABACDACA. Initially, there are four nodes that correspond to the characters of the

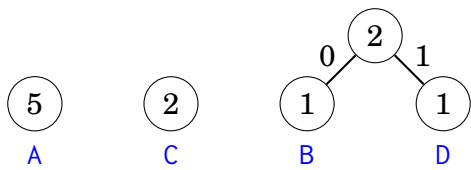
^{*2} D. A. Huffman discovered this method when solving a university course assignment and published the algorithm in 1952 [40].

string:

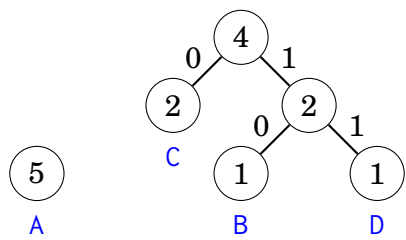


The node that represents character A has weight 5 because character A appears 5 times in the string. The other weights have been calculated in the same way.

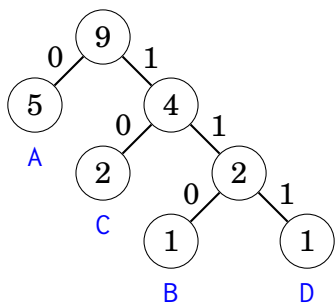
The first step is to combine the nodes that correspond to characters B and D, both with weight 1. The result is:



After this, the nodes with weight 2 are combined:



Finally, the two remaining nodes are combined:



Now all nodes are in the tree, so the code is ready. The following codewords can be read from the tree:

| character | codeword |
|-----------|----------|
| A | 0 |
| B | 110 |
| C | 10 |
| D | 111 |

第 7 章

Dynamic programming

Dynamic programming is a technique that combines the correctness of complete search and the efficiency of greedy algorithms. Dynamic programming can be applied if the problem can be divided into overlapping subproblems that can be solved independently.

There are two uses for dynamic programming:

- **Finding an optimal solution:** We want to find a solution that is as large as possible or as small as possible.
- **Counting the number of solutions:** We want to calculate the total number of possible solutions.

We will first see how dynamic programming can be used to find an optimal solution, and then we will use the same idea for counting the solutions.

Understanding dynamic programming is a milestone in every competitive programmer's career. While the basic idea is simple, the challenge is how to apply dynamic programming to different problems. This chapter introduces a set of classic problems that are a good starting point.

7.1 Coin problem

We first focus on a problem that we have already seen in Chapter 6: Given a set of coin values $\text{coins} = \{c_1, c_2, \dots, c_k\}$ and a target sum of money n , our task is to form the sum n using as few coins as possible.

In Chapter 6, we solved the problem using a greedy algorithm that always chooses the largest possible coin. The greedy algorithm works, for example, when

the coins are the euro coins, but in the general case the greedy algorithm does not necessarily produce an optimal solution.

Now is time to solve the problem efficiently using dynamic programming, so that the algorithm works for any coin set. The dynamic programming algorithm is based on a recursive function that goes through all possibilities how to form the sum, like a brute force algorithm. However, the dynamic programming algorithm is efficient because it uses *memoization* and calculates the answer to each subproblem only once.

Recursive formulation

The idea in dynamic programming is to formulate the problem recursively so that the solution to the problem can be calculated from solutions to smaller subproblems. In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required to form a sum x ?

Let $\text{solve}(x)$ denote the minimum number of coins required for a sum x . The values of the function depend on the values of the coins. For example, if $\text{coins} = \{1, 3, 4\}$, the first values of the function are as follows:

| | | |
|--------------------|-----|---|
| $\text{solve}(0)$ | $=$ | 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 |

For example, $\text{solve}(10) = 3$, because at least 3 coins are needed to form the sum 10. The optimal solution is $3 + 3 + 4 = 10$.

The essential property of solve is that its values can be recursively calculated from its smaller values. The idea is to focus on the *first* coin that we choose for the sum. For example, in the above scenario, the first coin can be either 1, 3 or 4. If we first choose coin 1, the remaining task is to form the sum 9 using the minimum number of coins, which is a subproblem of the original problem. Of course, the same applies to coins 3 and 4. Thus, we can use the following

recursive formula to calculate the minimum number of coins:

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

The base case of the recursion is $\text{solve}(0) = 0$, because no coins are needed to form an empty sum. For example,

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

Now we are ready to give a general recursive function that calculates the minimum number of coins needed to form a sum x :

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

First, if $x < 0$, the value is ∞ , because it is impossible to form a negative sum of money. Then, if $x = 0$, the value is 0, because no coins are needed to form an empty sum. Finally, if $x > 0$, the variable c goes through all possibilities how to choose the first coin of the sum.

Once a recursive function that solves the problem has been found, we can directly implement a solution in C++ (the constant INF denotes infinity):

```
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;
}
```

Still, this function is not efficient, because there may be an exponential number of ways to construct the sum. However, next we will see how to make the function efficient using a technique called memoization.

Using memoization

The idea of dynamic programming is to use **memoization** to efficiently calculate values of a recursive function. This means that the values of the function

are stored in an array after calculating them. For each parameter, the value of the function is calculated recursively only once, and after this, the value can be directly retrieved from the array.

In this problem, we use arrays

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

where $\text{ready}[x]$ indicates whether the value of $\text{solve}(x)$ has been calculated, and if it is, $\text{value}[x]$ contains this value. The constant N has been chosen so that all required values fit in the arrays.

Now the function can be efficiently implemented as follows:

```
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;
}
```

The function handles the base cases $x < 0$ and $x = 0$ as previously. Then the function checks from $\text{ready}[x]$ if $\text{solve}(x)$ has already been stored in $\text{value}[x]$, and if it is, the function directly returns it. Otherwise the function calculates the value of $\text{solve}(x)$ recursively and stores it in $\text{value}[x]$.

This function works efficiently, because the answer for each parameter x is calculated recursively only once. After a value of $\text{solve}(x)$ has been stored in $\text{value}[x]$, it can be efficiently retrieved whenever the function will be called again with the parameter x . The time complexity of the algorithm is $O(nk)$, where n is the target sum and k is the number of coins.

Note that we can also *iteratively* construct the array `value` using a loop that simply calculates all the values of `solve` for parameters $0 \dots n$:

```
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);
        }
    }
}

```

In fact, most competitive programmers prefer this implementation, because it is shorter and has lower constant factors. From now on, we also use iterative implementations in our examples. Still, it is often easier to think about dynamic programming solutions in terms of recursive functions.

Constructing a solution

Sometimes we are asked both to find the value of an optimal solution and to give an example how such a solution can be constructed. In the coin problem, for example, we can declare another array that indicates for each sum of money the first coin in an optimal solution:

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

Then, we can modify the algorithm as follows:

```

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;
        }
    }
}

```

After this, the following code can be used to print the coins that appear in an optimal solution for the sum n :

```

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

```

Counting the number of solutions

Let us now consider another version of the coin problem where our task is to calculate the total number of ways to produce a sum x using the coins. For example, if $\text{coins} = \{1, 3, 4\}$ and $x = 5$, there are a total of 6 ways:

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

Again, we can solve the problem recursively. Let $\text{solve}(x)$ denote the number of ways we can form the sum x . For example, if $\text{coins} = \{1, 3, 4\}$, then $\text{solve}(5) = 6$ and the recursive formula is

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

Then, the general recursive function is as follows:

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

If $x < 0$, the value is 0, because there are no solutions. If $x = 0$, the value is 1, because there is only one way to form an empty sum. Otherwise we calculate the sum of all values of the form $\text{solve}(x-c)$ where c is in coins .

The following code constructs an array `count` such that `count[x]` equals the value of $\text{solve}(x)$ for $0 \leq x \leq n$:

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

Often the number of solutions is so large that it is not required to calculate the exact number but it is enough to give the answer modulo m where, for example, $m = 10^9 + 7$. This can be done by changing the code so that all calculations are done modulo m . In the above code, it suffices to add the line

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

after the line

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

Now we have discussed all basic ideas of dynamic programming. Since dynamic programming can be used in many different situations, we will now go through a set of problems that show further examples about the possibilities of dynamic programming.


7.2 Longest increasing subsequence

Our first problem is to find the **longest increasing subsequence** in an array of n elements. This is a maximum-length sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element. For example, in the array

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

the longest increasing subsequence contains 4 elements:

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



Let $\text{length}(k)$ denote the length of the longest increasing subsequence that ends at position k . Thus, if we calculate all values of $\text{length}(k)$ where $0 \leq k \leq n-1$, we will find out the length of the longest increasing subsequence. For example, the values of the function for the above array are as follows:

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

For example, $\text{length}(6) = 4$, because the longest increasing subsequence that ends at position 6 consists of 4 elements.

To calculate a value of $\text{length}(k)$, we should find a position $i < k$ for which $\text{array}[i] < \text{array}[k]$ and $\text{length}(i)$ is as large as possible. Then we know that $\text{length}(k) = \text{length}(i) + 1$, because this is an optimal way to add $\text{array}[k]$ to a subsequence. However, if there is no such position i , then $\text{length}(k) = 1$, which means that the subsequence only contains $\text{array}[k]$.

Since all values of the function can be calculated from its smaller values, we can use dynamic programming. In the following code, the values of the function will be stored in an array `length`.

```
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);
        }
    }
}
```

This code works in $O(n^2)$ time, because it consists of two nested loops. However, it is also possible to implement the dynamic programming calculation more efficiently in $O(n \log n)$ time. Can you find a way to do this?

7.3 Paths in a grid

Our next problem is to find a path from the upper-left corner to the lower-right corner of an $n \times n$ grid, such that we only move down and right. Each square contains a positive integer, and the path should be constructed so that the sum of the values along the path is as large as possible.

The following picture shows an optimal path in a grid:

| | | | | |
|---|---|---|---|----|
| 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 |

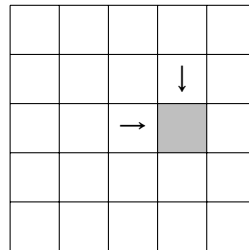
The sum of the values on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

Assume that the rows and columns of the grid are numbered from 1 to n , and $\text{value}[y][x]$ equals the value of square (y, x) . Let $\text{sum}(y, x)$ denote the maximum sum on a path from the upper-left corner to square (y, x) . Now $\text{sum}(n, n)$ tells us the maximum sum from the upper-left corner to the lower-right corner. For example, in the above grid, $\text{sum}(5, 5) = 67$.

We can recursively calculate the sums as follows:

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

The recursive formula is based on the observation that a path that ends at square (y, x) can come either from square $(y, x - 1)$ or square $(y - 1, x)$:



Thus, we select the direction that maximizes the sum. We assume that $\text{sum}(y, x) = 0$ if $y = 0$ or $x = 0$ (because no such paths exist), so the recursive formula also works when $y = 1$ or $x = 1$.

Since the function sum has two parameters, the dynamic programming array also has two dimensions. For example, we can use an array

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

and calculate the sums as follows:

```
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];
    }
}
```

The time complexity of the algorithm is $O(n^2)$.

In this section, we focus on the following problem: Given a list of weights $[w_1, w_2, \dots, w_n]$, determine all sums that can be constructed using the weights. For example, if the weights are $[1, 3, 3, 5]$, the following sums are possible:

[illegible]

To solve the problem, we focus on subproblems where we only use the first k weights to construct sums. Let $\text{possible}(x, k) = \text{true}$ if we can construct a sum x using the first k weights, and otherwise $\text{possible}(x, k) = \text{false}$. The values of the function can be recursively calculated as follows:

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

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

The following table shows all values of the function for the weights [1,3,3,5] (the symbol "X" indicates the true values):

[illegible]

After calculating those values, $\text{possible}(x, n)$ tells us whether we can construct a sum x using *all* weights.

Let W denote the total sum of the weights. The following $O(nW)$ time dynamic programming solution corresponds to the recursive function:

```
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];
    }
}
```

However, here is a better implementation that only uses a one-dimensional array $\text{possible}[x]$ that indicates whether we can construct a subset with sum x . The trick is to update the array from right to left for each new weight:

```
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;
    }
}
```

Note that the general idea presented here can be used in many knapsack problems. For example, if we are given objects with weights and values, we can determine for each weight sum the maximum value sum of a subset.

7.5 Edit distance

The **edit distance** or **Levenshtein distance**^{*1} is the minimum number of editing operations needed to transform a string into another string. The allowed editing operations are as follows:

- insert a character (e.g. ABC \rightarrow ABCA)
- remove a character (e.g. ABC \rightarrow AC)
- modify a character (e.g. ABC \rightarrow ADC)

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

For example, the edit distance between LOVE and MOVIE is 2, because we can first perform the operation LOVE \rightarrow MOVE (modify) and then the operation MOVE \rightarrow MOVIE (insert). This is the smallest possible number of operations, because it is clear that only one operation is not enough.

Suppose that we are given a string x of length n and a string y of length m , and we want to calculate the edit distance between x and y . To solve the problem, we define a function $\text{distance}(a, b)$ that gives the edit distance between prefixes $x[0 \dots a]$ and $y[0 \dots b]$. Thus, using this function, the edit distance between x and y equals $\text{distance}(n-1, m-1)$.

We can calculate values of distance as follows:

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

Here $\text{cost}(a, b) = 0$ if $x[a] = y[b]$, and otherwise $\text{cost}(a, b) = 1$. The formula considers the following ways to edit the string x :

- $\text{distance}(a, b-1)$: insert a character at the end of x
- $\text{distance}(a-1, b)$: remove the last character from x
- $\text{distance}(a-1, b-1)$: match or modify the last character of x

In the two first cases, one editing operation is needed (insert or remove). In the last case, if $x[a] = y[b]$, we can match the last characters without editing, and otherwise one editing operation is needed (modify).

The following table shows the values of distance in the example case:

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

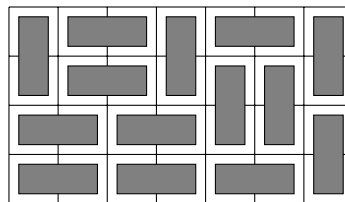
The lower-right corner of the table tells us that the edit distance between LOVE and MOVIE is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:

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

The last characters of LOVE and MOVIE are equal, so the edit distance between them equals the edit distance between LOV and MOVI. We can use one editing operation to remove the character I from MOVI. Thus, the edit distance is one larger than the edit distance between LOV and MOV, etc.

7.6 Counting tilings

Sometimes the states of a dynamic programming solution are more complex than fixed combinations of numbers. As an example, consider the problem of calculating the number of distinct ways to fill an $n \times m$ grid using 1×2 and 2×1 size tiles. For example, one valid solution for the 4×7 grid is



and the total number of solutions is 781.

The problem can be solved using dynamic programming by going through the grid row by row. Each row in a solution can be represented as a string that contains m characters from the set $\{\square, \sqcup, \sqsubset, \sqsupset\}$. For example, the above solution consists of four rows that correspond to the following strings:

- $\square \sqsubset \sqsubset \square \sqsubset \sqsubset \square$
- $\sqcup \sqsubset \sqcup \square \square \sqcup$
- $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \square$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup$

Let $\text{count}(k, x)$ denote the number of ways to construct a solution for rows $1 \dots k$ of the grid such that string x corresponds to row k . It is possible to use dynamic programming here, because the state of a row is constrained only by the state of

the previous row.

A solution is valid if row 1 does not contain the character \sqcup , row n does not contain the character \sqcap , and all consecutive rows are *compatible*. For example, the rows $\sqcup \square \square \sqcup \sqcap \sqcap \sqcup$ and $\square \square \square \square \sqcup \sqcup \sqcap$ are compatible, while the rows $\sqcap \square \square \sqcap \square \square \sqcap$ and $\square \square \square \square \square \sqcup$ are not compatible.

Since a row consists of m characters and there are four choices for each character, the number of distinct rows is at most 4^m . Thus, the time complexity of the solution is $O(n4^{2m})$ because we can go through the $O(4^m)$ possible states for each row, and for each state, there are $O(4^m)$ possible states for the previous row. In practice, it is a good idea to rotate the grid so that the shorter side has length m , because the factor 4^{2m} dominates the time complexity.

It is possible to make the solution more efficient by using a more compact representation for the rows. It turns out that it is sufficient to know which columns of the previous row contain the upper square of a vertical tile. Thus, we can represent a row using only characters \sqcap and \square , where \square is a combination of characters \sqcup , \square and \sqcap . Using this representation, there are only 2^m distinct rows and the time complexity is $O(n2^{2m})$.

As a final note, there is also a surprising direct formula for calculating the number of tilings^{*2}:

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

This formula is very efficient, because it calculates the number of tilings in $O(nm)$ time, but since the answer is a product of real numbers, a problem when using the formula is how to store the intermediate results accurately.

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

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。これは、どんなループを含み、何回ループが実行されるかといった解析を行います。ただし、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

まず、 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 |
|---|---|---|---|---|---|---|---|

これは、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 ポインターを使うことで $O(n)$ 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、各ステップでにおいて、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。そして、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、 $x=8$ を達成するための例を見てみましょう。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |
| ↑ | | ↑ | | | | | |

ここで、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 |
| | ↑ | ↑ | | | | | |

ただし、この状態でも 5 を加えると x を超えてしまうので、左のポインタを右に進めます。こうすると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は、まず、右ポインタの移動ステップ数に注目します。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 $O(n)$ ステップです。ここで、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、このアルゴリズムは $O(n)$ で動作する。

2SUM problem

他の問題を考えます。 n 個の数値からなる配列と 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。まず、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、対象の和が $x = 12$ である場合を考える。

| | | | | | | | |
|---|---|---|---|---|---|---|----|
| 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 |
| | ↑ | | | ↑ | | | |

ここで、左ポインタは再び右へ 1 ステップ移動します。右のポインタは動かないため、 $5 + 7 = 12$ 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 |
| | | ↑ | | ↑ | | | |

実行時間は、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、 $O(n \log n)$ となります。なお、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。これは、配列を繰り返し、配列の各値に対して、和 x をもたらす別の値を二分探索を行うことで行います。

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 Nearest smaller elements

Amortized analysis is often used to estimate the number of operations performed on a data structure. The operations may be distributed unevenly so that most operations occur during a certain phase of the algorithm, but the total number of the operations is limited.

^{*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]

As an example, consider the problem of finding for each array element the **nearest smaller element**, i.e., the first smaller element that precedes the element in the array. It is possible that no such element exists, in which case the algorithm should report this. Next we will see how the problem can be efficiently solved using a stack structure.

We go through the array from left to right and maintain a stack of array elements. At each array position, we remove elements from the stack until the top element is smaller than the current element, or the stack is empty. Then, we report that the top element is the nearest smaller element of the current element, or if the stack is empty, there is no such element. Finally, we add the current element to the stack.

As an example, consider the following array:

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

First, the elements 1, 3 and 4 are added to the stack, because each element is larger than the previous element. Thus, the nearest smaller element of 4 is 3, and the nearest smaller element of 3 is 1.

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

| | | | | |
|---|---|---|---|---|
| 1 | → | 3 | → | 4 |
|---|---|---|---|---|

The next element 2 is smaller than the two top elements in the stack. Thus, the elements 3 and 4 are removed from the stack, and then the element 2 is added to the stack. Its nearest smaller element is 1:

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

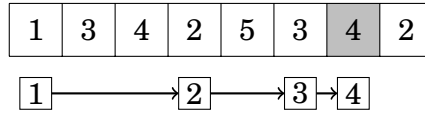
| | | |
|---|---|---|
| 1 | → | 2 |
|---|---|---|

Then, the element 5 is larger than the element 2, so it will be added to the stack, and its nearest smaller element is 2:

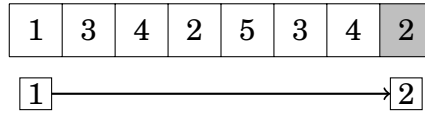
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 4 | 2 | 5 | 3 | 4 | 2 |
|---|---|---|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 1 | → | 2 | → | 5 |
|---|---|---|---|---|

After this, the element 5 is removed from the stack and the elements 3 and 4 are added to the stack:



Finally, all elements except 1 are removed from the stack and the last element 2 is added to the stack:



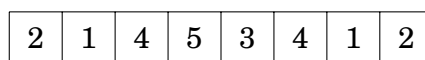
The efficiency of the algorithm depends on the total number of stack operations. If the current element is larger than the top element in the stack, it is directly added to the stack, which is efficient. However, sometimes the stack can contain several larger elements and it takes time to remove them. Still, each element is added *exactly once* to the stack and removed *at most once* from the stack. Thus, each element causes $O(1)$ stack operations, and the algorithm works in $O(n)$ time.

8.3 Sliding window minimum

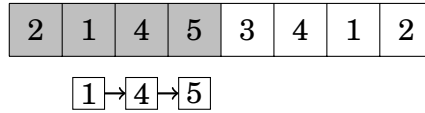
A **sliding window** is a constant-size subarray that moves from left to right through the array. At each window position, we want to calculate some information about the elements inside the window. In this section, we focus on the problem of maintaining the **sliding window minimum**, which means that we should report the smallest value inside each window.

The sliding window minimum can be calculated using a similar idea that we used to calculate the nearest smaller elements. We maintain a queue where each element is larger than the previous element, and the first element always corresponds to the minimum element inside the window. After each window move, we remove elements from the end of the queue until the last queue element is smaller than the new window element, or the queue becomes empty. We also remove the first queue element if it is not inside the window anymore. Finally, we add the new window element to the end of the queue.

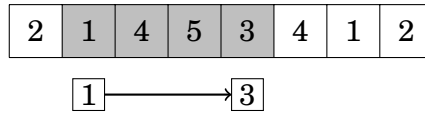
As an example, consider the following array:



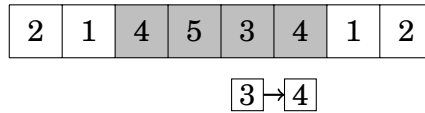
Suppose that the size of the sliding window is 4. At the first window position, the smallest value is 1:



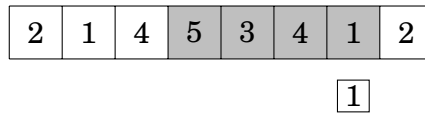
Then the window moves one step right. The new element 3 is smaller than the elements 4 and 5 in the queue, so the elements 4 and 5 are removed from the queue and the element 3 is added to the queue. The smallest value is still 1.



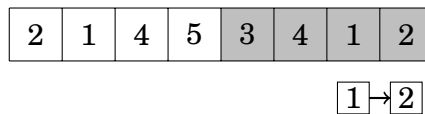
After this, the window moves again, and the smallest element 1 does not belong to the window anymore. Thus, it is removed from the queue and the smallest value is now 3. Also the new element 4 is added to the queue.



The next new element 1 is smaller than all elements in the queue. Thus, all elements are removed from the queue and it will only contain the element 1:



Finally the window reaches its last position. The element 2 is added to the queue, but the smallest value inside the window is still 1.



Since each array element is added to the queue exactly once and removed from the queue at most once, the algorithm works in $O(n)$ time.

第 9 章

Range queries

In this chapter, we discuss data structures that allow us to efficiently process range queries. In a **range query**, our task is to calculate a value based on a subarray of an array. Typical range queries are:

- $\text{sum}_q(a, b)$: calculate the sum of values in range $[a, b]$
- $\text{min}_q(a, b)$: find the minimum value in range $[a, b]$
- $\text{max}_q(a, b)$: find the maximum value in range $[a, b]$

For example, consider the range $[3, 6]$ in the following array:

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

In this case, $\text{sum}_q(3, 6) = 14$, $\text{min}_q(3, 6) = 1$ and $\text{max}_q(3, 6) = 6$.

A simple way to process range queries is to use a loop that goes through all array values in the range. For example, the following function can be used to process sum queries on an array:

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

This function works in $O(n)$ time, where n is the size of the array. Thus, we can process q queries in $O(nq)$ time using the function. However, if both n and q

are large, this approach is slow. Fortunately, it turns out that there are ways to process range queries much more efficiently.

9.1 Static array queries

We first focus on a situation where the array is *static*, i.e., the array values are never updated between the queries. In this case, it suffices to construct a static data structure that tells us the answer for any possible query.

Sum queries

We can easily process sum queries on a static array by constructing a **prefix sum array**. Each value in the prefix sum array equals the sum of values in the original array up to that position, i.e., the value at position k is $\text{sum}_q(0, k)$. The prefix sum array can be constructed in $O(n)$ time.

For example, consider the following array:

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

The corresponding prefix sum array is as follows:

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

Since the prefix sum array contains all values of $\text{sum}_q(0, k)$, we can calculate any value of $\text{sum}_q(a, b)$ in $O(1)$ time as follows:

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

By defining $\text{sum}_q(0, -1) = 0$, the above formula also holds when $a = 0$.

For example, consider the range $[3, 6]$:

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

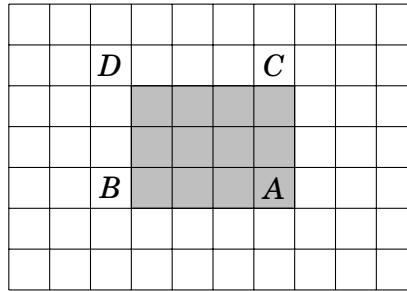
In this case $\text{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. This sum can be calculated from two values of the prefix sum array:

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

Thus, $\text{sum}_q(3, 6) = \text{sum}_q(0, 6) - \text{sum}_q(0, 2) = 27 - 8 = 19$.

It is also possible to generalize this idea to higher dimensions. For example, we can construct a two-dimensional prefix sum array that can be used to calculate the sum of any rectangular subarray in $O(1)$ time. Each sum in such an array corresponds to a subarray that begins at the upper-left corner of the array.

The following picture illustrates the idea:



The sum of the gray subarray can be calculated using the formula

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

where $S(X)$ denotes the sum of values in a rectangular subarray from the upper-left corner to the position of X .

Minimum queries

Minimum queries are more difficult to process than sum queries. Still, there is a quite simple $O(n \log n)$ time preprocessing method after which we can answer any minimum query in $O(1)$ time^{*1}. Note that since minimum and maximum queries can be processed similarly, we can focus on minimum queries.

The idea is to precalculate all values of $\min_q(a, b)$ where $b - a + 1$ (the length of the range) is a power of two. For example, for the array

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

the following values are calculated:

^{*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 | | | | | | |

The number of precalculated values is $O(n \log n)$, because there are $O(\log n)$ range lengths that are powers of two. The values can be calculated efficiently using the recursive formula

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

where $b - a + 1$ is a power of two and $w = (b - a + 1)/2$. Calculating all those values takes $O(n \log n)$ time.

After this, any value of $\min_q(a, b)$ can be calculated in $O(1)$ time as a minimum of two precalculated values. Let k be the largest power of two that does not exceed $b - a + 1$. We can calculate the value of $\min_q(a, b)$ using the formula

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

In the above formula, the range $[a, b]$ is represented as the union of the ranges $[a, a + k - 1]$ and $[b - k + 1, b]$, both of length k .

As an example, consider the range $[1, 6]$:

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

The length of the range is 6, and the largest power of two that does not exceed 6 is 4. Thus the range $[1, 6]$ is the union of the ranges $[1, 4]$ and $[3, 6]$:

| 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 |

Since $\min_q(1, 4) = 3$ and $\min_q(3, 6) = 1$, we conclude that $\min_q(1, 6) = 1$.

9.2 Binary indexed tree

A **binary indexed tree** or a **Fenwick tree**^{*2} can be seen as a dynamic variant of a prefix sum array. It supports two $O(\log n)$ time operations on an array: processing a range sum query and updating a value.

The advantage of a binary indexed tree is that it allows us to efficiently update array values between sum queries. This would not be possible using a prefix sum array, because after each update, it would be necessary to build the whole prefix sum array again in $O(n)$ time.

Structure

Even if the name of the structure is a binary indexed *tree*, it is usually represented as an array. In this section we assume that all arrays are one-indexed, because it makes the implementation easier.

Let $p(k)$ denote the largest power of two that divides k . We store a binary indexed tree as an array *tree* such that

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

i.e., each position k contains the sum of values in a range of the original array whose length is $p(k)$ and that ends at position k . For example, since $p(6) = 2$, $\text{tree}[6]$ contains the value of $\text{sum}_q(5, 6)$.

For example, consider the following array:

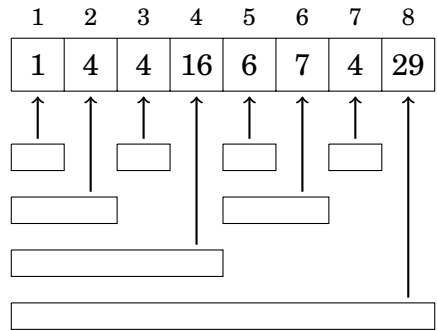
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 4 | 8 | 6 | 1 | 4 | 2 |

The corresponding binary indexed tree is as follows:

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

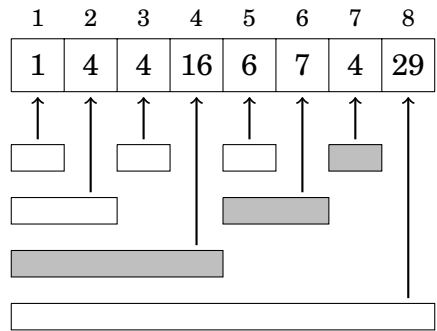
The following picture shows more clearly how each value in the binary indexed tree corresponds to a range in the original array:

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



Using a binary indexed tree, any value of $\text{sum}_q(1, k)$ can be calculated in $O(\log n)$ time, because a range $[1, k]$ can always be divided into $O(\log n)$ ranges whose sums are stored in the tree.

For example, the range $[1, 7]$ consists of the following ranges:



Thus, we can calculate the corresponding sum as follows:

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

To calculate the value of $\text{sum}_q(a, b)$ where $a > 1$, we can use the same trick that we used with prefix sum arrays:

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

Since we can calculate both $\text{sum}_q(1, b)$ and $\text{sum}_q(1, a - 1)$ in $O(\log n)$ time, the total time complexity is $O(\log n)$.

Then, after updating a value in the original array, several values in the binary indexed tree should be updated. For example, if the value at position 3 changes, the sums of the following ranges change:

9.3 Segment tree

A **segment tree**^{*3} is a data structure that supports two operations: processing a range query and updating an array value. Segment trees can support sum queries, minimum and maximum queries and many other queries so that both operations work in $O(\log n)$ time.

Compared to a binary indexed tree, the advantage of a segment tree is that it is a more general data structure. While binary indexed trees only support sum queries^{*4}, segment trees also support other queries. On the other hand, a segment tree requires more memory and is a bit more difficult to implement.

Structure

A segment tree is a binary tree such that the nodes on the bottom level of the tree correspond to the array elements, and the other nodes contain information needed for processing range queries.

In this section, we assume that the size of the array is a power of two and zero-based indexing is used, because it is convenient to build a segment tree for such an array. If the size of the array is not a power of two, we can always append extra elements to it.

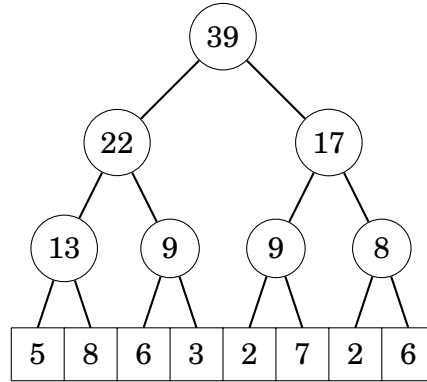
We will first discuss segment trees that support sum queries. As an example, consider the following array:

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

The corresponding segment tree is as follows:

^{*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.

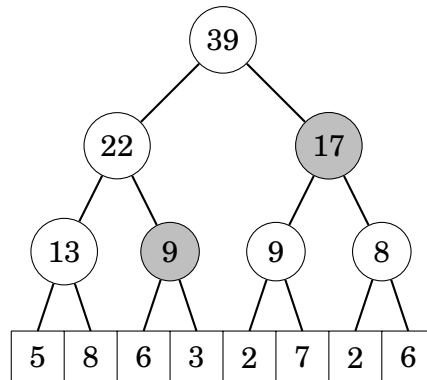


Each internal tree node corresponds to an array range whose size is a power of two. In the above tree, the value of each internal node is the sum of the corresponding array values, and it can be calculated as the sum of the values of its left and right child node.

It turns out that any range $[a, b]$ can be divided into $O(\log n)$ ranges whose values are stored in tree nodes. For example, consider the range $[2, 7]$:

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

Here $\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$. In this case, the following two tree nodes correspond to the range:

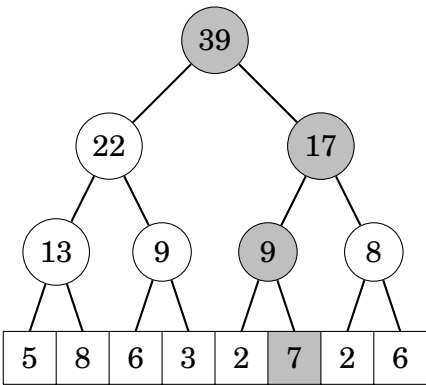


Thus, another way to calculate the sum is $9 + 17 = 26$.

When the sum is calculated using nodes located as high as possible in the tree, at most two nodes on each level of the tree are needed. Hence, the total number of nodes is $O(\log n)$.

After an array update, we should update all nodes whose value depends on the updated value. This can be done by traversing the path from the updated array element to the top node and updating the nodes along the path.

The following picture shows which tree nodes change if the array value 7 changes:

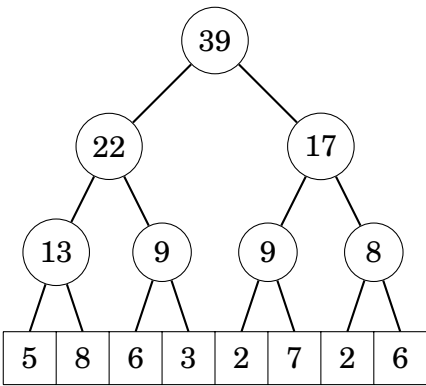


The path from bottom to top always consists of $O(\log n)$ nodes, so each update changes $O(\log n)$ nodes in the tree.

Implementation

We store a segment tree as an array of $2n$ elements where n is the size of the original array and a power of two. The tree nodes are stored from top to bottom: $\text{tree}[1]$ is the top node, $\text{tree}[2]$ and $\text{tree}[3]$ are its children, and so on. Finally, the values from $\text{tree}[n]$ to $\text{tree}[2n - 1]$ correspond to the values of the original array on the bottom level of the tree.

For example, the segment tree



is stored as follows:

| | | | | | | | | | | | | | | |
|----|----|----|----|---|---|---|---|---|----|----|----|----|----|----|
| 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 |

Using this representation, the parent of $\text{tree}[k]$ is $\text{tree}[\lfloor k/2 \rfloor]$, and its children are $\text{tree}[2k]$ and $\text{tree}[2k + 1]$. Note that this implies that the position of a node

is even if it is a left child and odd if it is a right child.

The following function calculates the value of $\text{sum}_q(a, b)$:

```
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;
}
```

The function maintains a range that is initially $[a + n, b + n]$. Then, at each step, the range is moved one level higher in the tree, and before that, the values of the nodes that do not belong to the higher range are added to the sum.

The following function increases the array value at position k by x :

```
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];
    }
}
```

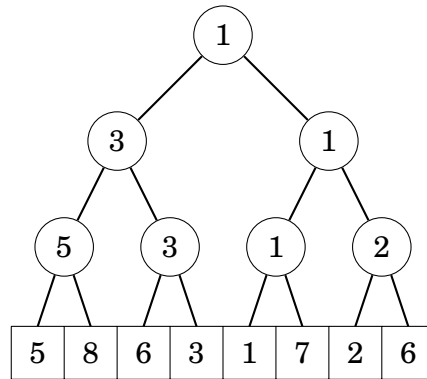
First the function updates the value at the bottom level of the tree. After this, the function updates the values of all internal tree nodes, until it reaches the top node of the tree.

Both the above functions work in $O(\log n)$ time, because a segment tree of n elements consists of $O(\log n)$ levels, and the functions move one level higher in the tree at each step.

Other queries

Segment trees can support all range queries where it is possible to divide a range into two parts, calculate the answer separately for both parts and then efficiently combine the answers. Examples of such queries are minimum and maximum, greatest common divisor, and bit operations and, or and xor.

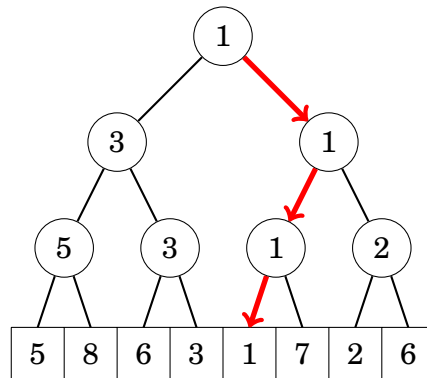
For example, the following segment tree supports minimum queries:



In this case, every tree node contains the smallest value in the corresponding array range. The top node of the tree contains the smallest value in the whole array. The operations can be implemented like previously, but instead of sums, minima are calculated.

The structure of a segment tree also allows us to use binary search for locating array elements. For example, if the tree supports minimum queries, we can find the position of an element with the smallest value in $O(\log n)$ time.

For example, in the above tree, an element with the smallest value 1 can be found by traversing a path downwards from the top node:



9.4 Additional techniques

Index compression

A limitation in data structures that are built upon an array is that the elements are indexed using consecutive integers. Difficulties arise when large indices are needed. For example, if we wish to use the index 10^9 , the array should contain 10^9 elements which would require too much memory.

However, we can often bypass this limitation by using **index compression**, where the original indices are replaced with indices 1, 2, 3, etc. This can be done if we know all the indices needed during the algorithm beforehand.

The idea is to replace each original index x with $c(x)$ where c is a function that compresses the indices. We require that the order of the indices does not change, so if $a < b$, then $c(a) < c(b)$. This allows us to conveniently perform queries even if the indices are compressed.

For example, if the original indices are 555, 10^9 and 8, the new indices are:

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

Range updates

So far, we have implemented data structures that support range queries and updates of single values. Let us now consider an opposite situation, where we should update ranges and retrieve single values. We focus on an operation that increases all elements in a range $[a, b]$ by x .

Surprisingly, we can use the data structures presented in this chapter also in this situation. To do this, we build a **difference array** whose values indicate the differences between consecutive values in the original array. Thus, the original array is the prefix sum array of the difference array. For example, consider the following array:

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

The difference array for the above array is as follows:

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

For example, the value 2 at position 6 in the original array corresponds to the sum $3 - 2 + 4 - 3 = 2$ in the difference array.

The advantage of the difference array is that we can update a range in the original array by changing just two elements in the difference array. For example, if we want to increase the original array values between positions 1 and 4 by 5, it suffices to increase the difference array value at position 1 by 5 and decrease the

value at position 5 by 5. The result is as follows:

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

More generally, to increase the values in range $[a, b]$ by x , we increase the value at position a by x and decrease the value at position $b + 1$ by x . Thus, it is only needed to update single values and process sum queries, so we can use a binary indexed tree or a segment tree.

A more difficult problem is to support both range queries and range updates. In Chapter 28 we will see that even this is possible.

The first bit in a signed representation is the sign of the number (0 for nonnegative numbers and 1 for negative numbers), and the remaining $n - 1$ bits contain the magnitude of the number. **Two's complement** is used, which means that the opposite number of a number is calculated by first inverting all the bits in the number, and then increasing the number by one.

For example, the bit representation of the int number -43 is

1111111111111111111111111111010101.

In an unsigned representation, only nonnegative numbers can be used, but the upper bound for the values is larger. An unsigned variable of n bits can contain any integer between 0 and $2^n - 1$. For example, in C++, an unsigned int variable can contain any integer between 0 and $2^{32} - 1$.

There is a connection between the representations: a signed number $-x$ equals an unsigned number $2^n - x$. For example, the following code shows that the signed number $x = -43$ equals the unsigned number $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

If a number is larger than the upper bound of the bit representation, the number will overflow. In a signed representation, the next number after $2^{n-1} - 1$ is -2^{n-1} , and in an unsigned representation, the next number after $2^n - 1$ is 0. For example, consider the following code:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Initially, the value of x is $2^{31} - 1$. This is the largest value that can be stored in an int variable, so the next number after $2^{31} - 1$ is -2^{31} .

10.2 Bit operations

And operation

The **and** operation $x \& y$ produces a number that has one bits in positions where both x and y have one bits. For example, $22 \& 26 = 18$, because

$$\begin{array}{r} 10110 \quad (22) \\ \& \quad 11010 \quad (26) \\ \hline = \quad 10010 \quad (18) \end{array}$$

Using the and operation, we can check if a number x is even because $x \& 1 = 0$ if x is even, and $x \& 1 = 1$ if x is odd. More generally, x is divisible by 2^k exactly when $x \& (2^k - 1) = 0$.

Or operation

The **or** operation $x \mid y$ produces a number that has one bits in positions where at least one of x and y have one bits. For example, $22 \mid 26 = 30$, because

$$\begin{array}{r} 10110 \quad (22) \\ \mid \quad 11010 \quad (26) \\ \hline = \quad 11110 \quad (30) \end{array}$$

Xor operation

The **xor** operation $x \wedge y$ produces a number that has one bits in positions where exactly one of x and y have one bits. For example, $22 \wedge 26 = 12$, because

$$\begin{array}{r} 10110 \quad (22) \\ \wedge \quad 11010 \quad (26) \\ \hline = \quad 01100 \quad (12) \end{array}$$

Not operation

The **not** operation $\sim x$ produces a number where all the bits of x have been inverted. The formula $\sim x = -x - 1$ holds, for example, $\sim 29 = -30$.

The result of the not operation at the bit level depends on the length of the bit representation, because the operation inverts all bits. For example, if the numbers are 32-bit int numbers, the result is as follows:

$$\begin{array}{rcl} x & = & 29 \quad 000000000000000000000000011101 \\ \sim x & = & -30 \quad 111111111111111111111111100010 \end{array}$$

Bit shifts

The left bit shift $x \ll k$ appends k zero bits to the number, and the right bit shift $x \gg k$ removes the k last bits from the number. For example, $14 \ll 2 = 56$, because 14 and 56 correspond to 1110 and 111000. Similarly, $49 \gg 3 = 6$, because 49 and 6 correspond to 110001 and 110.

Note that $x \ll k$ corresponds to multiplying x by 2^k , and $x \gg k$ corresponds to dividing x by 2^k rounded down to an integer.

Applications

A number of the form $1 \ll k$ has a one bit in position k and all other bits are zero, so we can use such numbers to access single bits of numbers. In particular, the k th bit of a number is one exactly when $x \& (1 \ll k)$ is not zero. The following code prints the bit representation of an `int` number x :

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

It is also possible to modify single bits of numbers using similar ideas. For example, the formula $x \mid (1 \ll k)$ sets the k th bit of x to one, the formula $x \& \sim(1 \ll k)$ sets the k th bit of x to zero, and the formula $x \wedge (1 \ll k)$ inverts the k th bit of x .

The formula $x \& (x - 1)$ sets the last one bit of x to zero, and the formula $x \& -x$ sets all the one bits to zero, except for the last one bit. The formula $x \mid (x - 1)$ inverts all the bits after the last one bit. Also note that a positive number x is a power of two exactly when $x \& (x - 1) = 0$.

Additional functions

The g++ compiler provides the following functions for counting bits:

- `__builtin_clz(x)`: the number of zeros at the beginning of the number
- `__builtin_ctz(x)`: the number of zeros at the end of the number
- `__builtin_popcount(x)`: the number of ones in the number

- `__builtin_parity(x)`: the parity (even or odd) of the number of ones

The functions can be used as follows:

```
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
```

While the above functions only support `int` numbers, there are also long long versions of the functions available with the suffix `ll`.

10.3 Representing sets

Every subset of a set $\{0, 1, 2, \dots, n-1\}$ can be represented as an n bit integer whose one bits indicate which elements belong to the subset. This is an efficient way to represent sets, because every element requires only one bit of memory, and set operations can be implemented as bit operations.

For example, since `int` is a 32-bit type, an `int` number can represent any subset of the set $\{0, 1, 2, \dots, 31\}$. The bit representation of the set $\{1, 3, 4, 8\}$ is

000000000000000000000000100011010,

which corresponds to the number $2^8 + 2^4 + 2^3 + 2^1 = 282$.

Set implementation

The following code declares an `int` variable x that can contain a subset of $\{0, 1, 2, \dots, 31\}$. After this, the code adds the elements 1, 3, 4 and 8 to the set and prints the size of the set.

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

Then, the following code prints all elements that belong to the set:

```
for (int i = 0; i < 32; i++) {
```

```

    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8

```

Set operations

Set operations can be implemented as follows as bit operations:

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

For example, the following code first constructs the sets $x = \{1, 3, 4, 8\}$ and $y = \{3, 6, 8, 9\}$, and then constructs the set $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$:

```

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

```

Iterating through subsets

The following code goes through the subsets of $\{0, 1, \dots, n-1\}$:

```

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

```

The following code goes through the subsets with exactly k elements:

```

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

```

The following code goes through the subsets of a set x :

```

int b = 0;
do {

```

```
// process subset b
} while (b=(b-x)&x);
```

10.4 Bit optimizations

Many algorithms can be optimized using bit operations. Such optimizations do not change the time complexity of the algorithm, but they may have a large impact on the actual running time of the code. In this section we discuss examples of such situations.

Hamming distances

The **Hamming distance** $\text{hamming}(a, b)$ between two strings a and b of equal length is the number of positions where the strings differ. For example,

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

Consider the following problem: Given a list of n bit strings, each of length k , calculate the minimum Hamming distance between two strings in the list. For example, the answer for $[00111, 01101, 11110]$ is 2, because

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

A straightforward way to solve the problem is to go through all pairs of strings and calculate their Hamming distances, which yields an $O(n^2k)$ time algorithm. The following function can be used to calculate distances:

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

However, if k is small, we can optimize the code by storing the bit strings as integers and calculating the Hamming distances using bit operations. In particular, if $k \leq 32$, we can just store the strings as `int` values and use the

following function to calculate distances:

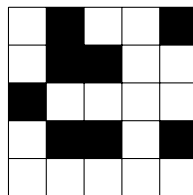
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

In the above function, the xor operation constructs a bit string that has one bits in positions where a and b differ. Then, the number of bits is calculated using the `__builtin_popcount` function.

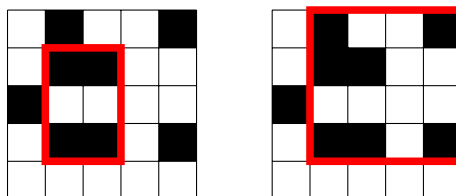
To compare the implementations, we generated a list of 10000 random bit strings of length 30. Using the first approach, the search took 13.5 seconds, and after the bit optimization, it only took 0.5 seconds. Thus, the bit optimized code was almost 30 times faster than the original code.

Counting subgrids

As another example, consider the following problem: Given an $n \times n$ grid whose each square is either black (1) or white (0), calculate the number of subgrids whose all corners are black. For example, the grid



contains two such subgrids:



There is an $O(n^3)$ time algorithm for solving the problem: go through all $O(n^2)$ pairs of rows and for each pair (a, b) calculate the number of columns that contain a black square in both rows in $O(n)$ time. The following code assumes that `color[y][x]` denotes the color in row y and column x :

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



```
}

```

Then, those columns account for $\text{count}(\text{count} - 1)/2$ subgrids with black corners, because we can choose any two of them to form a subgrid.

To optimize this algorithm, we divide the grid into blocks of columns such that each block consists of N consecutive columns. Then, each row is stored as a list of N -bit numbers that describe the colors of the squares. Now we can process N columns at the same time using bit operations. In the following code, $\text{color}[y][k]$ represents a block of N colors as bits.

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

The resulting algorithm works in $O(n^3/N)$ time.

We generated a random grid of size 2500×2500 and compared the original and bit optimized implementation. While the original code took 29.6 seconds, the bit optimized version only took 3.1 seconds with $N = 32$ (int numbers) and 1.7 seconds with $N = 64$ (long long numbers).

10.5 Dynamic programming

Bit operations provide an efficient and convenient way to implement dynamic programming algorithms whose states contain subsets of elements, because such states can be stored as integers. Next we discuss examples of combining bit operations and dynamic programming.

Optimal selection

As a first example, consider the following problem: We are given the prices of k products over n days, and we want to buy each product exactly once. However, we are allowed to buy at most one product in a day. What is the minimum total price? For example, consider the following scenario ($k = 3$ and $n = 8$):

| | 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 |

In this scenario, the minimum total price is 5:

| | 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 |

Let $\text{price}[x][d]$ denote the price of product x on day d . For example, in the above scenario $\text{price}[2][3] = 7$. Then, let $\text{total}(S, d)$ denote the minimum total price for buying a subset S of products by day d . Using this function, the solution to the problem is $\text{total}(\{0 \dots k-1\}, n-1)$.

First, $\text{total}(\emptyset, d) = 0$, because it does not cost anything to buy an empty set, and $\text{total}(\{x\}, 0) = \text{price}[x][0]$, because there is one way to buy one product on the first day. Then, the following recurrence can be used:

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

This means that we either do not buy any product on day d or buy a product x that belongs to S . In the latter case, we remove x from S and add the price of x to the total price.

The next step is to calculate the values of the function using dynamic programming. To store the function values, we declare an array

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

where K and N are suitably large constants. The first dimension of the array corresponds to a bit representation of a subset.

First, the cases where $d = 0$ can be processed as follows:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Then, the recurrence translates into the following code:

```

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]);
            }
        }
    }
}

```

The time complexity of the algorithm is $O(n2^k k)$.

From permutations to subsets

Using dynamic programming, it is often possible to change an iteration over permutations into an iteration over subsets^{*1}. The benefit of this is that $n!$, the number of permutations, is much larger than 2^n , the number of subsets. For example, if $n = 20$, then $n! \approx 2.4 \cdot 10^{18}$ and $2^n \approx 10^6$. Thus, for certain values of n , we can efficiently go through the subsets but not through the permutations.

As an example, consider the following problem: There is an elevator with maximum weight x , and n people with known weights who want to get from the ground floor to the top floor. What is the minimum number of rides needed if the people enter the elevator in an optimal order?

For example, suppose that $x = 10$, $n = 5$ and the weights are as follows:

| person | weight |
|--------|--------|
| 0 | 2 |
| 1 | 3 |
| 2 | 3 |
| 3 | 5 |
| 4 | 6 |

In this case, the minimum number of rides is 2. One optimal order is {0,2,3,1,4}, which partitions the people into two rides: first {0,2,3} (total weight 10), and then {1,4} (total weight 9).

^{*1} This technique was introduced in 1962 by M. Held and R. M. Karp [34].

The problem can be easily solved in $O(n!n)$ time by testing all possible permutations of n people. However, we can use dynamic programming to get a more efficient $O(2^n n)$ time algorithm. The idea is to calculate for each subset of people two values: the minimum number of rides needed and the minimum weight of people who ride in the last group.

Let $\text{weight}[p]$ denote the weight of person p . We define two functions: $\text{rides}(S)$ is the minimum number of rides for a subset S , and $\text{last}(S)$ is the minimum weight of the last ride. For example, in the above scenario

$$\text{rides}(\{1,3,4\}) = 2 \quad \text{and} \quad \text{last}(\{1,3,4\}) = 5,$$

because the optimal rides are $\{1,4\}$ and $\{3\}$, and the second ride has weight 5. Of course, our final goal is to calculate the value of $\text{rides}(\{0 \dots n-1\})$.

We can calculate the values of the functions recursively and then apply dynamic programming. The idea is to go through all people who belong to S and optimally choose the last person p who enters the elevator. Each such choice yields a subproblem for a smaller subset of people. If $\text{last}(S \setminus p) + \text{weight}[p] \leq x$, we can add p to the last ride. Otherwise, we have to reserve a new ride that initially only contains p .

To implement dynamic programming, we declare an array

```
pair<int,int> best[1<<N];
```

that contains for each subset S a pair $(\text{rides}(S), \text{last}(S))$. We set the value for an empty group as follows:

```
best[0] = {1,0};
```

Then, we can fill the array as follows:

```
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);
}
}
}

```

Note that the above loop guarantees that for any two subsets S_1 and S_2 such that $S_1 \subset S_2$, we process S_1 before S_2 . Thus, the dynamic programming values are calculated in the correct order.

Counting subsets

Our last problem in this chapter is as follows: Let $X = \{0 \dots n-1\}$, and each subset $S \subset X$ is assigned an integer $\text{value}[S]$. Our task is to calculate for each S

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A],$$

i.e., the sum of values of subsets of S .

For example, suppose that $n = 3$ and the values are as follows:

- $\text{value}[\emptyset] = 3$
- $\text{value}[\{0\}] = 1$
- $\text{value}[\{1\}] = 4$
- $\text{value}[\{0, 1\}] = 5$
- $\text{value}[\{2\}] = 5$
- $\text{value}[\{0, 2\}] = 1$
- $\text{value}[\{1, 2\}] = 3$
- $\text{value}[\{0, 1, 2\}] = 3$

In this case, for example,

$$\begin{aligned} \text{sum}(\{0, 2\}) &= \text{value}[\emptyset] + \text{value}[\{0\}] + \text{value}[\{2\}] + \text{value}[\{0, 2\}] \\ &= 3 + 1 + 5 + 1 = 10. \end{aligned}$$

Because there are a total of 2^n subsets, one possible solution is to go through all pairs of subsets in $O(2^{2n})$ time. However, using dynamic programming, we can solve the problem in $O(2^n n)$ time. The idea is to focus on sums where the elements that may be removed from S are restricted.

Let $\text{partial}(S, k)$ denote the sum of values of subsets of S with the restriction that only elements $0 \dots k$ may be removed from S . For example,

$$\text{partial}(\{0, 2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0, 2\}],$$

because we may only remove elements $0 \dots 1$. We can calculate values of sum using values of partial , because

$$\text{sum}(S) = \text{partial}(S, n - 1).$$

The base cases for the function are

$$\text{partial}(S, -1) = \text{value}[S],$$

because in this case no elements can be removed from S . Then, in the general case we can use the following recurrence:

$$\text{partial}(S, k) = \begin{cases} \text{partial}(S, k - 1) & k \notin S \\ \text{partial}(S, k - 1) + \text{partial}(S \setminus \{k\}, k - 1) & k \in S \end{cases}$$

Here we focus on the element k . If $k \in S$, we have two options: we may either keep k in S or remove it from S .

There is a particularly clever way to implement the calculation of sums. We can declare an array

```
int sum[1<<N];
```

that will contain the sum of each subset. The array is initialized as follows:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

Then, we can fill the array as follows:

```
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)];
    }
}
```

This code calculates the values of $\text{partial}(S, k)$ for $k = 0 \dots n - 1$ to the array sum . Since $\text{partial}(S, k)$ is always based on $\text{partial}(S, k - 1)$, we can reuse the array sum , which yields a very efficient implementation.

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

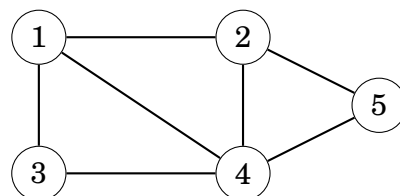
多くの競技プログラミング問題は、問題をグラフの問題としてモデル化し、適切なグラフアルゴリズムを用いることで解決することができます。グラフの典型的な例として、街を道路でつなぐネットワークとみなし最短の経路を求める問題などあります。しかし、コンテストではグラフが問題の中に隠れていて、それを発見することが難しいことも多々あります。

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

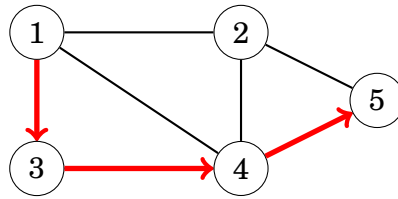
グラフ - **graph** はノード - **node** と辺 - **edge** から構成されます。本書では以降、変数 n はグラフのノード数、変数 m は辺の数を表します。ノードには $1, 2, v, \dots, n$ の整数を用いて表します (訳註: 0-indexed ではありません)。

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) がある。パスの長さは、その中に含まれる辺の数です。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。

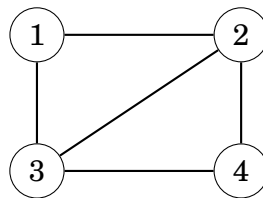


最初と最後のノードが同じものは**閉路** (サイクル, Cycle) と呼ばれます。例えば、上のグラフは $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ という閉路を含みます。

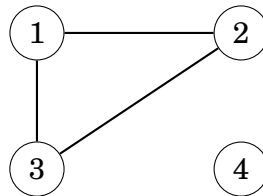
各ノードの出現回数がせいぜい 1 回であるとき、パスは**単純**と呼ばれます。

連結 - Connectivity

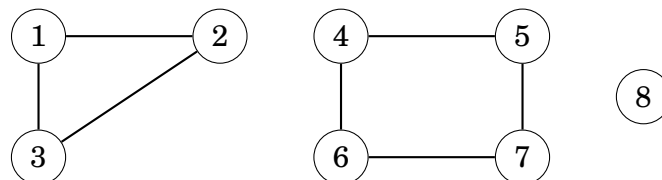
グラフは、ある任意の 2 つのノード間にパスが存在する場合、**連結**といいます。例えば、次のようなグラフは連結と呼ばれます。



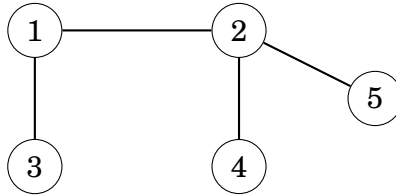
次のグラフは、ノード 4 から他のノードに行くことができないので、連結ではないグラフです (非連結)。



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。

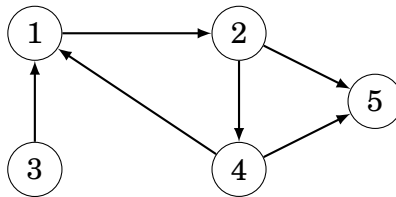


木 (tree) は、 n 個のノードと $n-1$ 個の辺からなる連結グラフのことです。木の任意の 2 つのノード間には一本のパスが存在します。例えば、次のグラフは木である。



有向と無向 - Edge directions

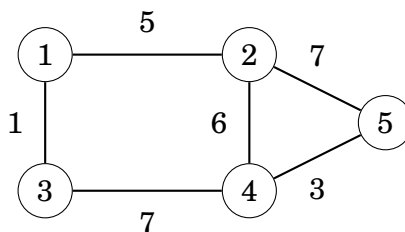
グラフの辺が決まった方向にしか移動できない場合、**有向 (directed)** と呼ばれます。例えば、次のようなグラフは有向です。



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

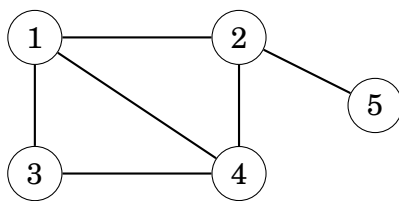
重み付きグラフでは、各辺に**重み**が割り当てられています。重みは辺の長さのようなものです。重み付きグラフの例を次に示します。



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

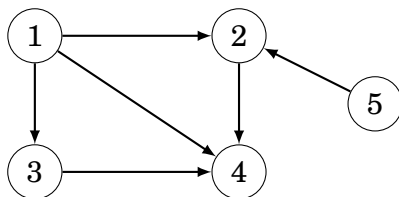
2つのノードは、その間にエッジがある場合、**隣接 (neighbors, adjacent)** であるといいます。**次数 (degree)** は、ノードの隣接するノードの数です。例えば、次のグラフでは、ノード 2 の近傍は 1、4、5 なので、その次数は 3 です。



グラフに含まれる頂点の次数の和は常に $2m$ (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ある。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。

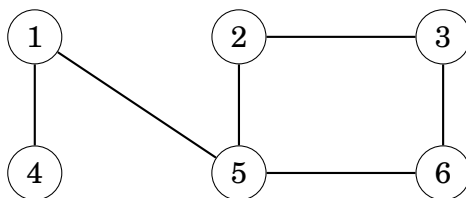
有向グラフでにおいて各頂点には入次数と出次数があります。**入次数 (indegree)** とはそのノードで終わる辺の数、**出次数 (outdegree)** そのノードで始まる辺の数です。例えば、次のグラフでは、ノード 2 の入次数は 2、ノード 2 の出次数は 1 です。



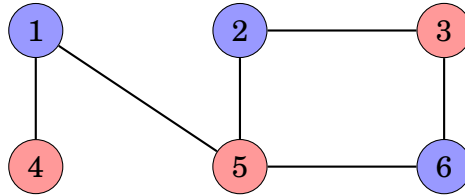
グラフの着色 - Colorings

グラフの色付け (**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。

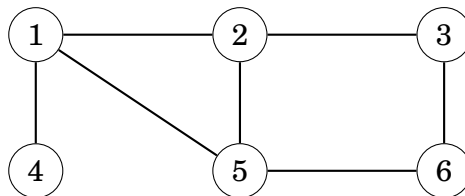
例を示します。



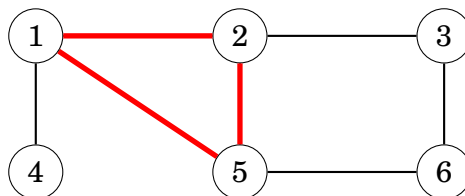
このグラフは次のように着色でき、二部グラフです。



次のグラフはどうでしょうか？

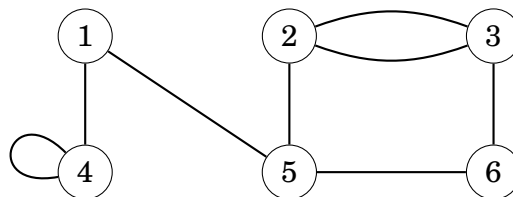


は 2 分割ではありません。なぜなら、以下の閉路を 2 色で色塗りできないためです。



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。

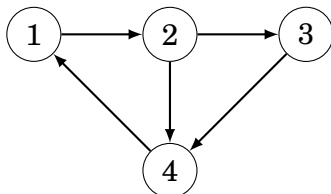
隣接リスト形式 - Adjacency list representation

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 はグラフの頂点の数です。例えば次のグラフを考えます。



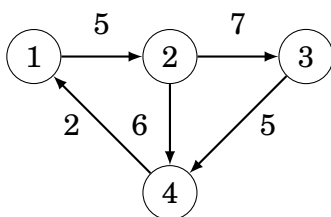
これは隣接リストで次のように表現できます。

```
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。

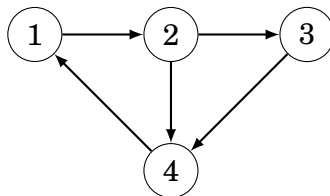
```
for (auto u : adj[s]) {
    // process node u
}
```

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];
```

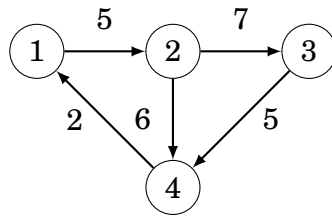
ここで、 $adj[a][b]$ は、グラフにノード a からノード b への辺が含まれるかをしめし、 $adj[a][b] = 1$ なら辺があることを、 $adj[a][b] = 0$ なら辺がないことを示します。例えば次のグラフを考えます。



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。If the graph is weighted, the adjacency matrix representation can be extended so that the matrix contains the weight of the edge if the edge exists. Using this representation, the graph



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 の要素を含んで通常はそのほとんどが 0 であることである。このため、グラフが大きい場合はこの表現は適しません。

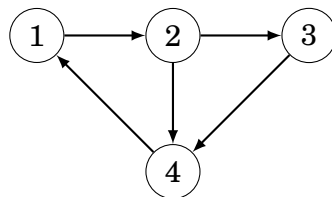
エッジリスト形式 - Edge list representation

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便に便利ですが、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。



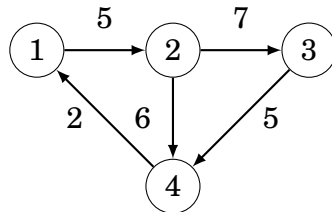
これは次のように表現できます。

```
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

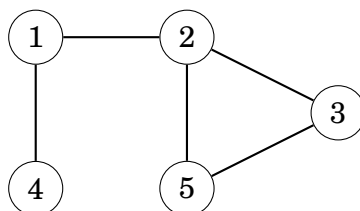
この章では、深さ優先探索 (DFS) と幅優先探索 (BFS) という 2 つの基本的なグラフアルゴリズムについて説明します。どちらのアルゴリズムもグラフの開始ノードが与えられ、その開始ノードからすべての到達可能なノードを訪問し、両者はこの訪問順序が異なります。

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、新しいノードが見つかる限り、常にグラフ内の単一経路をたどります。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。

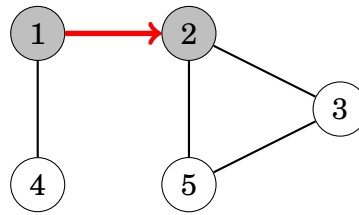
例

次のグラフを深さ優先探索で処理する方法を考えます。

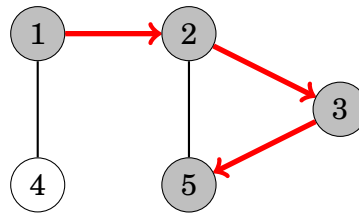


グラフのどのノードから検索を始めても良いですが、ここではノード 1 から検索を始めます。

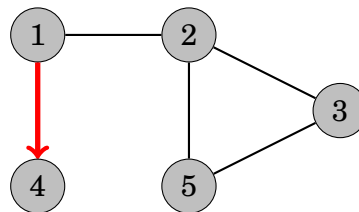
まず、ノード 2 を探索します。



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ため、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);
    }
}

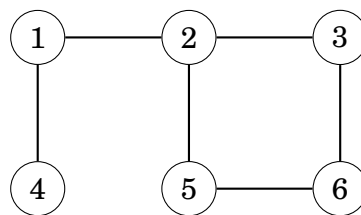
```

12.2 幅優先探索 - Breadth-first search

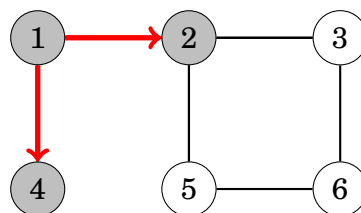
幅優先探索 - Breadth-first search (BFS) は、開始ノードからの距離順が小さい順にノードを訪問していきます。つまり、BFS を用いれば、始点ノードから他のすべてのノードまでの距離を計算することができます (訳註: これは DFS でも可能です)。ただし、BFS は深さ優先探索よりも実装が複雑になります。BFS は、ノードを深さごとに見ていきます。まず、開始ノードからの距離が 1 であるノードを探索し、次に距離が 2 であるノードを探索し、というようにすべてのノードが訪問されるまで続けられます。

例

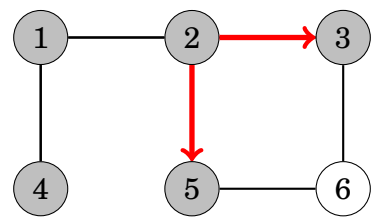
次のようなグラフに対して、幅優先探索がどのように処理されるかを考えてみよう。



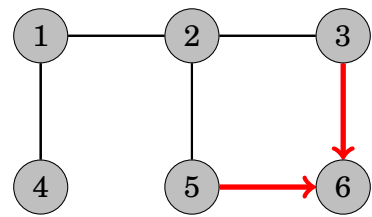
ノード 1 から探索を開始したとします。最初に、ノード 1 から直接つながっている 1 つの辺で到達可能なすべてのノードを探索します。



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 が辺の数、 m が頂点とします。

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 よりも実装が困難になります。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。

以下のコードでは、グラフが隣接リストとして格納され、以下のデータ構造を保持することを想定しています。

データ構造:

```
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る。新しいノードは常にキューの末尾に追加され、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 `texttt{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);
    }
}
```

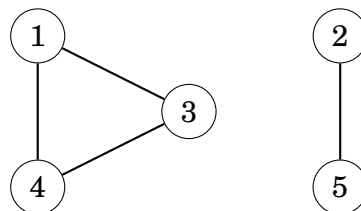
12.3 応用 - Applications

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るが、実際には `DFS` の方が実装が容易であるため、`DFS` を選択するのが良いでしょう。以下の応用例では、グラフは無向であると仮定する。

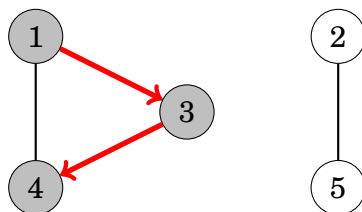
連結かのチェック - Connectivity check

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。

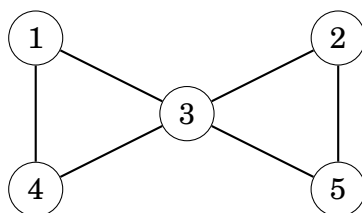


この結果、すべてのノードが訪問されなかったので、このグラフは連結されていないと結論づけられます。このあと、探索されていないノードからさらに新しい深さ優先探索を開始することにより、グラフのすべての連結成分を見つけることもできます。

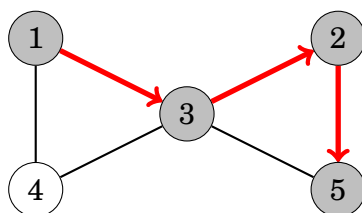
閉路の検出 - Finding cycles

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、グラフはサイクル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。

After moving from node 2 to node 5 we notice that the neighbor 3 of node 5 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

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

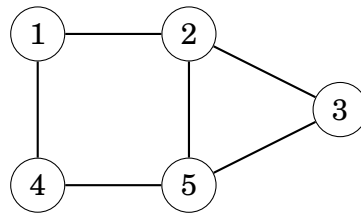
ノードとエッジの数を単純に計算することである。ある連結のグラフに c 個のノードがあり、閉路がなければ、ちょうど $c-1$ 個のエッジを含むはず（つまり、木でなければならない）。 c 個以上の辺があれば、この連結成分には必ず閉路が含まれます。

二部グラフチェック - Bipartiteness check

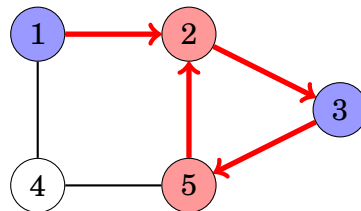
グラフを同じ色のノードが隣接しないようにのノードが 2 色で着色できる場合 2 部グラフです。グラフの探索アルゴリズムを用いて、グラフが 2 分割かどうかを調べるのは非常に簡単です。

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 2 分割できません。そうでなければ、グラフは 2 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 2 部グラフにはできません。

重要な点は、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。例えば、複数の都市があり、それぞれをつなぐ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)

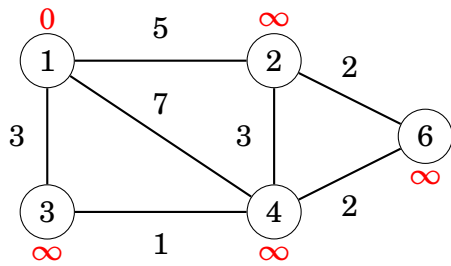
Bellman – Ford algorithm^{*1} はあるノードから開始して全てのノードへの最短経路を計算します。このアルゴリズムは負のコストとなる閉路を持たない全てのグラフで利用できます。もし、負のコストとなる閉路が存在する場合、その検出を行います。

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

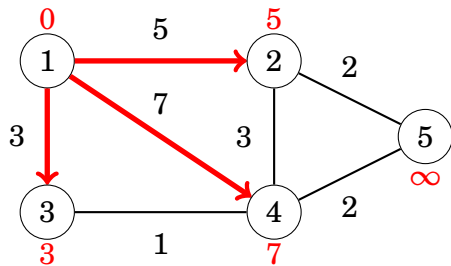
次の図で Bellman – Ford アルゴリズムの動作を説明します。

^{*1} The algorithm is named after R. E. Bellman and L. R. Ford who published it independently in 1958 and 1956, respectively [5, 24].

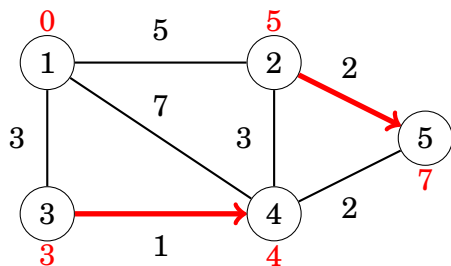


各ノードは距離を持ちます。開始ノードの距離は 0 と、それ以外の距離は INF です。

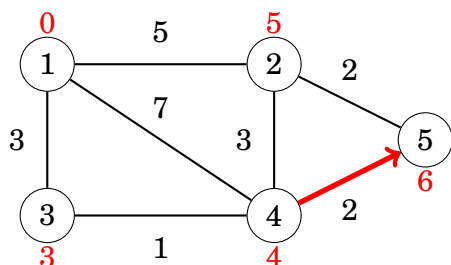
このアルゴリズムでは、あるノードから隣接するノードの距離を縮めることのできる辺を調べます。まず、ノード 1 から始めます。全ての隣接ノードへの距離は (初期値が INF なので) 縮まります。



次に 2 → 5 と 3 → 4 の 2 つの辺が距離を縮めます。

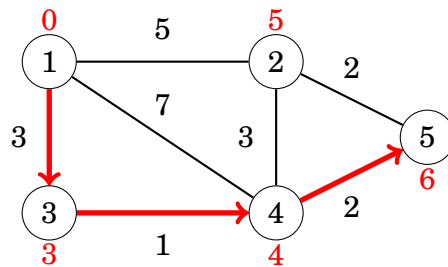


最後にもう 1 つ更新できます。



これ以上は距離を縮めることはできません。つまり、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

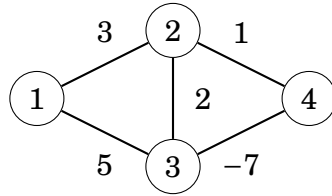
ノード x からグラフの全ノードまでの最短距離を求める実装を以下に示します。この実装は、グラフが (a, b, w) というタプルのリスト `edges` で表現されているとします。それぞれの要素は辺を表し、ノード a からノード b 向きに重み w の辺とします。Bellman – Ford は $n-1$ ラウンドで構成されます。各ラウンドはグラフのすべてのエッジを調べ、隣接ノードへの距離を縮めようと試みます。実装では、始点 x からグラフの全ノードまでの距離を格納する配列 `distance` を定義します。なお、定数 `INF` は、無限遠を表します。

```
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、 $n-1$ ラウンドよりも早く求められるのが普通でしょう。そのため、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るとより効率的なアルゴリズムになるでしょう。

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。



グラフに長さ -4 の負の閉路 $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ があるとしましょう。

グラフに負がある場合、無限回にそのサイクルに関係するノードの距離は小さくなっていきます。つまり、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ことになります。そしてこれは、開始ノードに関係なく、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、より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

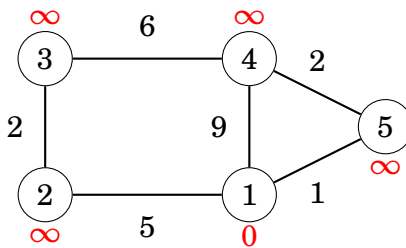
Dijkstra's 法 ^{*2} は、ベルマン-フォードと同様に、始点ノードからグラフの全ノードまでの最短経路を求める手法です。ダイクストラ法は、より大規模なグラフの処理に使用できます。注意すべき点として、このアルゴリズムはグラフ内に負の重みのエッジが存在しないことが必須の条件です。

^{*2} E. W. Dijkstra published the algorithm in 1959 [14]

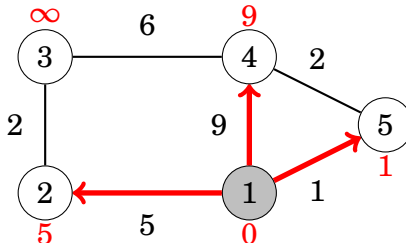
ベルマンフォードと同様に、ダイクストラ法は各ノードまでの距離を維持して、探索中に距離を更新していきます。ダイクストラ法では、グラフに負のエッジがないことを利用して、各エッジをただ一度だけ処理するので高速に動作します。

例

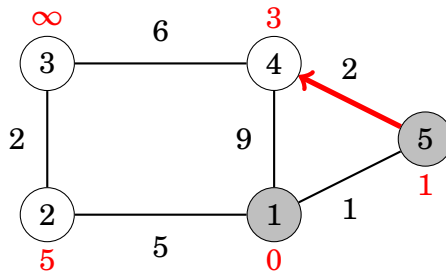
次のグラフで、ノード1を始点とするとき、Dijkstra のアルゴリズムがどのように働くかをみていきます。



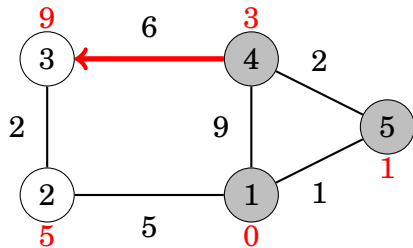
初期値はベルマン-フォードと同様に、開始ノードまでの距離は 0、他のすべてのノードまでの距離は INF とします。各ステップにおいて、Dijkstra のアルゴリズムは、まだ処理されていないノードの中から、その距離が最も小さいものをいずれか選択する。アルゴリズムが開始した際に選択される最初のノードは、距離 0 のノード 1 です。ノードが選択されると、アルゴリズムはそのノードを始点とするすべての辺をしらべ、その先のノードの距離を更新できるならば更新します。



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。

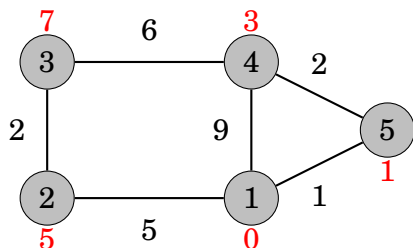


次はノード 4 であり、ノード 3 までの距離が 9 になりました。



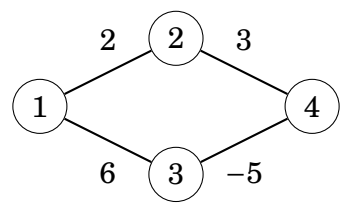
ダイクストラ法で注目する特徴は、あるノードの処理を開始するときはいつでも、その距離が最終的なものであることです。例えば、この時点では、距離 0、1、3 がノード 1、5、4 への最終的な距離である。

このように、処理を続け、最終的な距離 は以下のようにになります。



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 → 3 → 4 で、その長さは 1 である。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 → 2 → 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ません。

実装

以下に実装を示します。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、 $adj[a]$ 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ノードを含む優先度付きキュー (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。初期状態では、 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は、ノードへの距離をマイナスで保持しています。C++ のデフォルトの優先度付きキューは最大要素を返すので、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。また注意すべき点として、このキューには同じノードの処理待ちが入ることがありますが、距離が最小のもののみ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

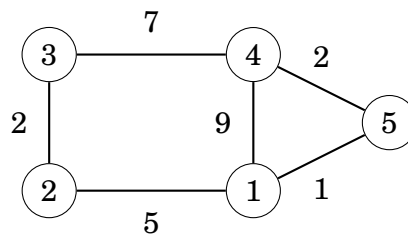
このアルゴリズムでは、ノード間の距離を含む 2 次元の配列を保持する。まず、ノード間の直接のエッジのみを用いて距離を計算し、この後、アルゴリズムがパス中の中間ノードを用いて距離を縮める。例

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た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、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

^{*4} The algorithm is named after R. W. Floyd and S. Warshall who published it independently in 1962 [23, 70].

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 |

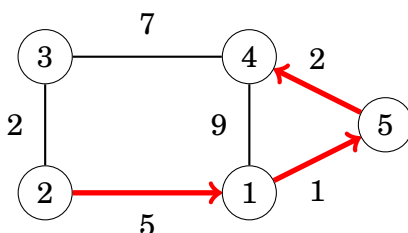
3 ラウンド目では、ノード 3 が新しい中間ノードとします。ノード 2 と 4 の間に新しいパスができます。

| | 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 |

例えば、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。

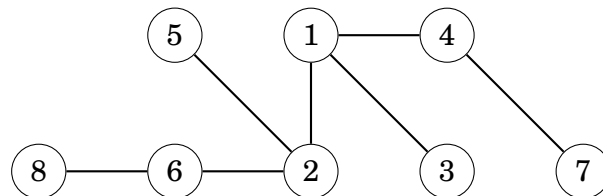
この実装は単純なので、グラフの最短経路を 1 つだけ見つける必要がある場合でも、このアルゴリズムは使用できます。ただし、このアルゴリズムが使えるのは、グラフが非常に小さく、3 乗の時間計算量で十分速くなる場合です。

第 14 章

木のアлゴリズム - Tree algorithms

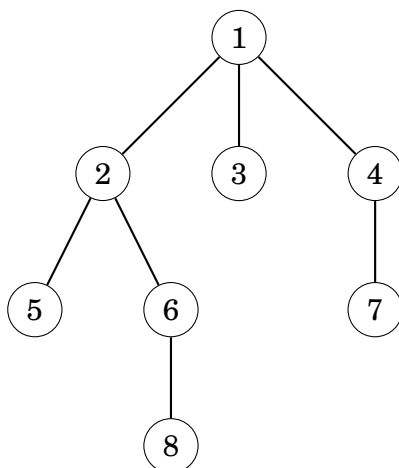
木は、 n 個のノードと $n-1$ 個のエッジからなる連結かつ閉路を含まないグラフです。木から辺を 1 つ取り除くと 2 つの要素に分かれ、辺を加えると 1 つの閉路ができます。また、木の任意の 2 つのノード間には常に一本のパスが存在します。例えば、8 個のノードと 7 個のエッジから構成されている木の例を示します。

For example, the following tree consists of 8 nodes and 7 edges:



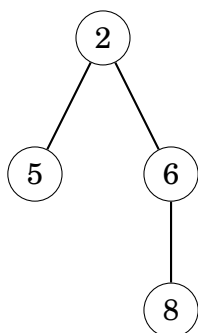
木の葉(リーフ)は、次数が 1 のノード、つまり、隣接ノードが 1 つしかないノードです。上図の葉は、ノード 3, 5, 7, 8 です。

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 木の走査 - Graph Traversal

一般的なグラフの走査アルゴリズムは、木のノードを走査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走査は一般のグラフの走査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 が与えられる. パラメータ e の目的は, まだ訪問していないノードにのみ移動するためです. 最初に次の様に呼ぶことでノード 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 は存在しません) 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

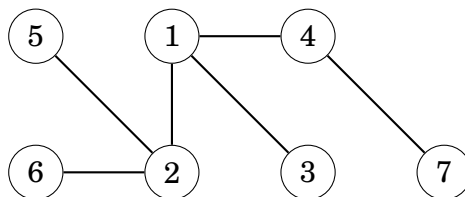
動的計画法は、木の走査中に何らかの情報を計算するために用いることができる。例えば、根付き木において各ノードの部分木のノード数、そのノードから葉までの最長経路の長さといったものを $O(n)$ 時間で計算できます。

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。

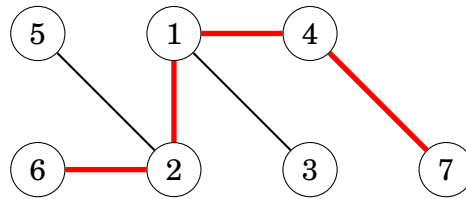
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

14.2 直径 - Diameter

木の**直径**とはその木における最長の 2 ノード間の距離です。例を示します。



この木の直径は次のパスの通り 4 です。

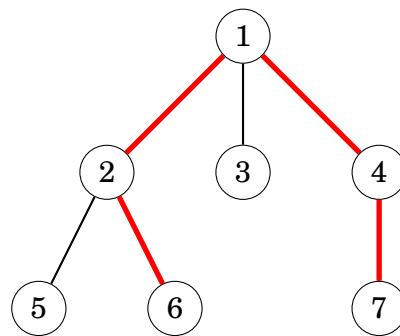


半径の経路は複数存在する可能性があります。例えば上記の例ならノード 6 をノード 5 に置き換えると、長さ 4 の別のパスを得ることができます。

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。こうするとそれぞれの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ます。このアルゴリズムはこのアプローチをとります。根付き木のすべてのパスには最高点があるということです。最高点とは、そのパスに属する最も高いノードです。各ノードについてそのノードを最高点とす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 が直径に相当する経路上の最高点である。



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、上の木では、 $1 \rightarrow 2 \rightarrow 6$ という経路があるので $\text{toLeaf}(1)=2$ 、 $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ という経路があるので $\text{maxLength}(1)=4$ となります。この場合、 $\text{maxLength}(1)$ は直径と等しくなる。

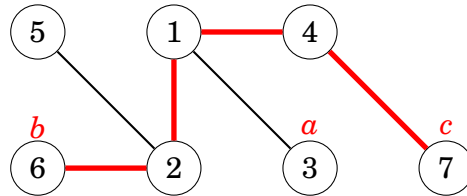
例えば上の図の木では $1 \rightarrow 2 \rightarrow 6$ があるため $\text{toLeaf}(1)=2$ となり、 $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ があるので $\text{maxLength}(1)=4$ となります。この例では $\text{maxLength}(1)$ が直径と

なります。

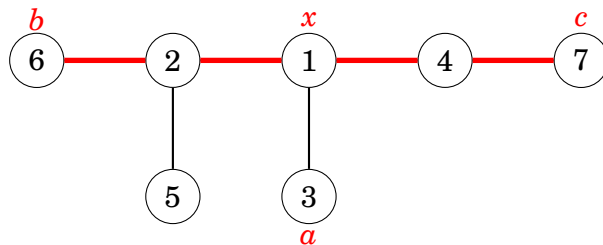
動的計画法を用いて上記の値を全ノードについて $O(n)$ 時間で計算することができます。まず、 $\text{toLeaf}(x)$ を計算するために、 x の子ノードを調べ、 $\text{toLeaf}(c)$ が最大である子 c を選んで、この値に 1 を足します。次に、 $\text{maxLength}(x)$ を計算するために、 $\text{toLeaf}(a) + \text{toLeaf}(b)$ の和が最大となるような異なる二つの子 a と b を選び、この和に 2 を足します。

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つ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。次に、 b から最も遠いノード c を見つける。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 を次の様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様に描いてみます。

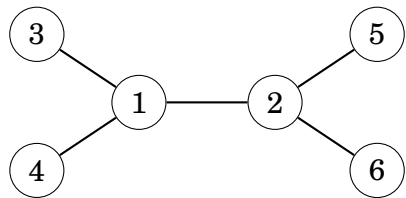


ノード x は、ノード a からの経路が直径に対応する経路に合流するノードとします。 a から最も遠いノードは b 、ノード c 、あるいはノード x から少なくとも同じだけ遠い他のノードです。したがって、このノードは直径に対応するパスの終点として常に有効な候補となります。(訳注: a から辿るとこの表現をした際の水平の線に必ず当たります。なので 1 回目の BFS でそこから最も遠い距離を探すことはいずれかの水平な線の候補の端点を示し、そこから BFS を再度行えば同じ様にもう一方の端点を探せます)

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 $O(n)$ 時間で解くことができます。



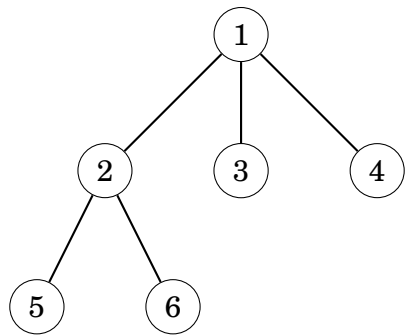
を始点とするパスの最大長を $\text{maxLength}(x)$ とする。 x . 例えば、上の木では、 $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ という経路があるので、 $\text{maxLength}(4)=3$ である。以下は、その値の完全な表である。

$\text{maxLength}(x)$ を x を始点とするパスの最大長とします。上図の木では $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ というパスがあるので、 $\text{maxLength}(4) = 3$ です。

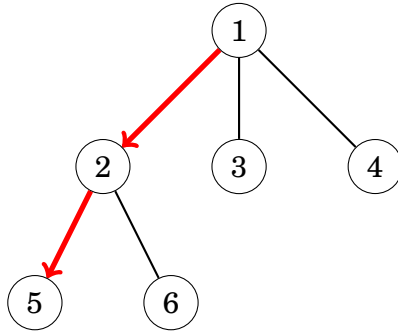
これを表にします。

| node 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。

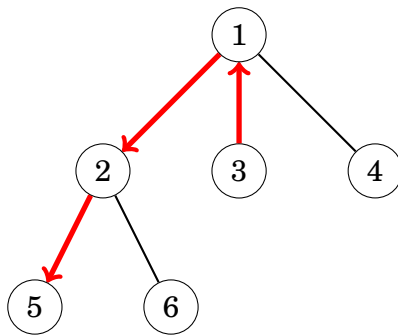


各 x について、その子のノードの中で最長のパスを持つノードを探します。例えば、ノード 1 からの最長の経路はその子 2 を通ることがわかります。

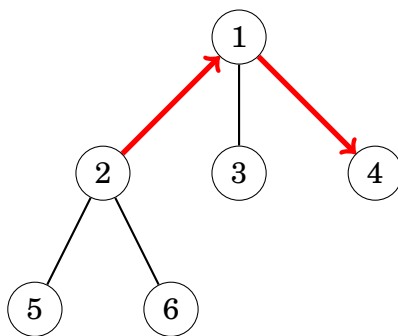


これは前回と同様に動的計画法で $O(n)$ で解くことができます。例えば、ノード 3 からの最長経路はその親 1 を通る。This part is easy to solve in $O(n)$ time, because we can use dynamic programming as we have done previously.

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。まず次の様に定義します。

- $\text{maxLength}_1(x)$: 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。

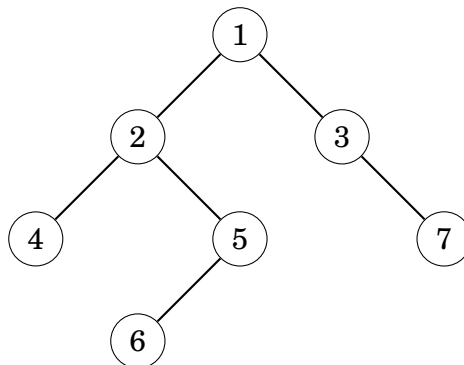
例えば、上記のグラフでは、 $1 \rightarrow 2 \rightarrow 5$ のパスを用いて、 $\text{maxLength}_1(1) = 2$ となり $\text{maxLength}_2(1) = 1$ パス $1 \rightarrow 3$ を使用する。最後に、 $\text{maxLength}_1(p)$ に対応するパスが x を通る場合、最大長は $\text{maxLength}_2(p)+1$ であり、それ以外の場合は最大長は $\text{maxLength}_1(p)+1$ であることを conclude する。

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。

次に、 $\text{maxLength}_1(p)$ 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、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。二分木のすべてのノードは、0 個、1 個、または 2 個の子を持と言えます。例を示します。



The nodes of a binary tree have three natural orderings that correspond to different ways to recursively traverse the tree:

なる方法に対応している。

- **preorder:** まずルート进行处理し、次に左のサブツリーをトラバースし、そして右のサブツリーをトラバースする。
- **順序:** まず左のサブツリーを走査し、次にルートを処理し、そして右のサブツリーを走査する。
- **ポストオーダー:** まず左のサブツリーをトラバースし、次に右のサブツリーをトラバースし、次にルートを処理する

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 and in-order 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。



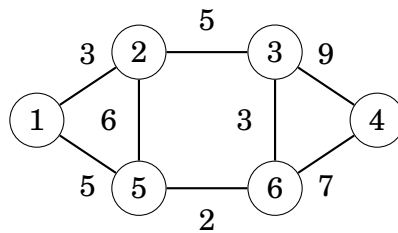
どちらの木も pre-order [1,2] で post-order is [2,1] です。このため、どちらかに確定させることはできません。

第 15 章

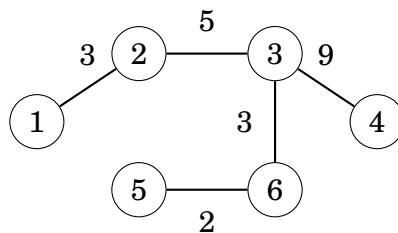
全域木 - Spanning trees

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、木は連結していてサイクルは存在しません。木を構成するにはいくつかの方法があります。

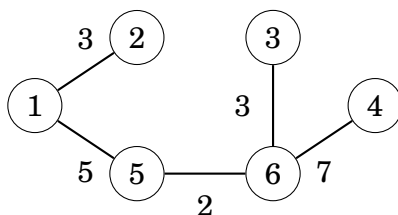
次のようなグラフを考えてみましょう。



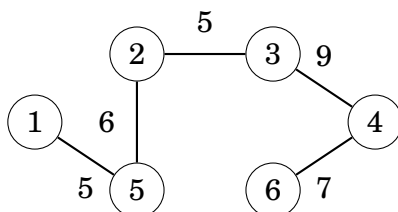
例えば、1 つの全域木は次のようになります。



全域木の重さとは、エッジの重みの合計です。例えば、上記の全域木の重みは、 $3 + 5 + 9 + 3 + 2 = 22$ 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。説明上、最小全域木だけに絞って説明を行いますが、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。

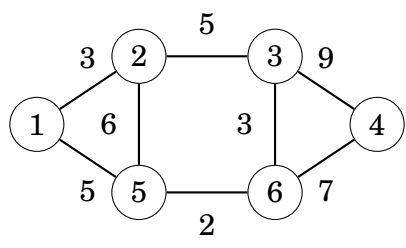
15.1 クラスカル法 - Kruskal's algorithm

Kruskal のアルゴリズム^{*1} は次のように動作します。まず、グラフのノードだけを持ち、エッジを含みません。次に、このアルゴリズムは、重みが小さい順にエッジを調べ、サイクル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いる。木にエッジを追加する時、その2つの成分を結合します。最後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

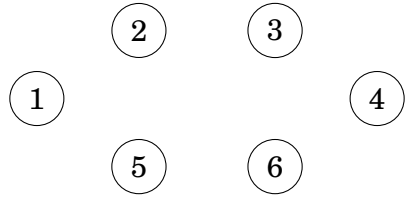
次のグラフを考えます。



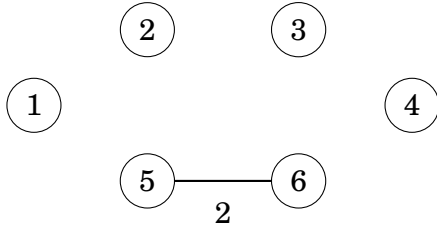
最初に全てのエッジを重みが小さい順にソートします。

| 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 |

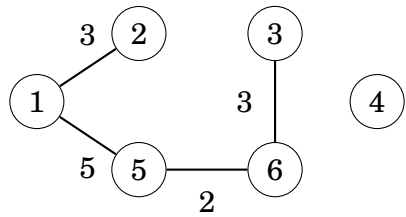
リストを上から処理し、各辺が2つの別の連結成分を結合するなら、その辺を接続します。



まず、5-6 の処理は {5} と {6} を連結し、{5,6} とします。



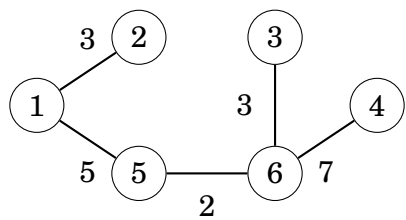
同じように、1-2, 3-6, 1-5 が接続されます。



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。

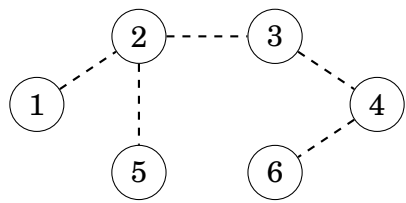
最後に、4-6 が接続されます。



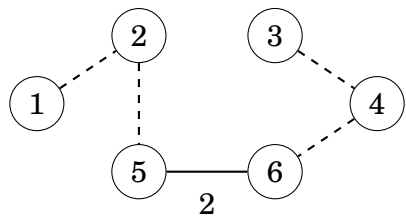
これで全ての点が同じ連結成分に含まれるので処理を終えます。これにより、重さが $2+3+3+5+7=20$ の最小全域木ができました。

なぜこれで良いか？

なぜこのアルゴリズムが有効なのか？ つまり、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、これによって最小のスパニングツリーが生成されます。つまり、重さが小さな順に辺を木に加えることも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、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。ただし、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。 $Union-Find$ 構造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。

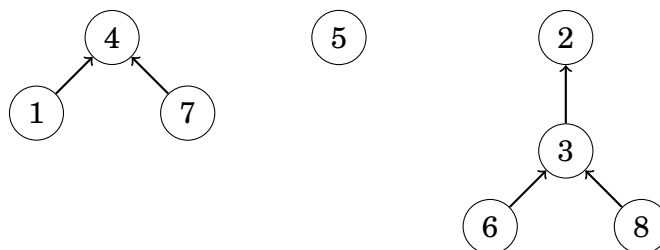
15.2 Union-find 構造 - Union-find structure

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}

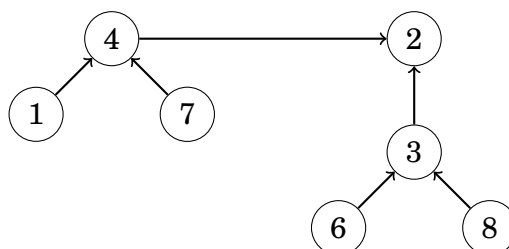
構造

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。例えば、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6の代表は要素2です。さて、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。そして、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 構造の計算量は、集合をどのように結合するか依存します。集合のサイズは小さい方の集合代表をと大きい方の集合の代表につなげばよいのである(同じ大きさの集合であれば、任意に選択することも可能である)。この方法を用いると、どのパスの長さも $O(\log n)$ となり、対応する鎖をたどることで任意の要素の代表を効率的に求めることができます。

Implementation

The union-find structure can be implemented using arrays. In the following implementation, the array `link` contains for each element the next element in the chain or the element itself if it is a representative, and the array `size` indicates for each representative the size of the corresponding set.

Initially, each element belongs to a separate set:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

The function `find` returns the representative for an element x . The representative can be found by following the chain that begins at x .

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

The function `same` checks whether elements a and b belong to the same set. This can easily be done by using the function `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

The function `unite` joins the sets that contain elements a and b (the elements have to be in different sets). The function first finds the representatives of the sets and then connects the smaller set to the larger set.

```
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;
}
```

The time complexity of the function `find` is $O(\log n)$ assuming that the length of each chain is $O(\log n)$. In this case, the functions `same` and `unite` also work in $O(\log n)$ time. The function `unite` makes sure that the length of each chain is $O(\log n)$ by connecting the smaller set to the larger set.

15.3 Prim's algorithm

Prim's algorithm^{*3} is an alternative method for finding a minimum spanning tree. The algorithm first adds an arbitrary node to the tree. After this, the algorithm always chooses a minimum-weight edge that adds a new node to the tree. Finally, all nodes have been added to the tree and a minimum spanning

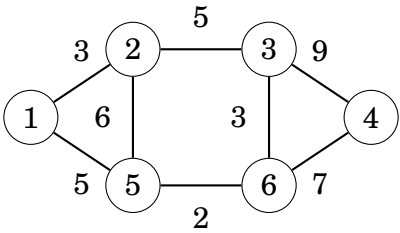
^{*3} The algorithm is named after R. C. Prim who published it in 1957 [54]. However, the same algorithm was discovered already in 1930 by V. Jarník.

tree has been found.

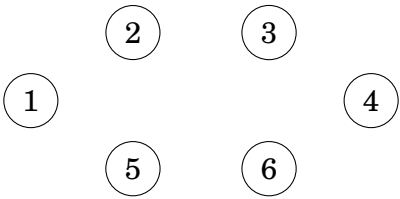
Prim’s algorithm resembles Dijkstra’s algorithm. The difference is that Dijkstra’s algorithm always selects an edge whose distance from the starting node is minimum, but Prim’s algorithm simply selects the minimum weight edge that adds a new node to the tree.

Example

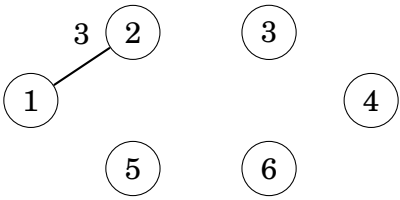
Let us consider how Prim’s algorithm works in the following graph:



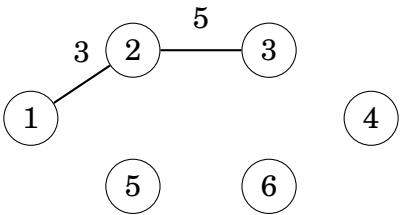
Initially, there are no edges between the nodes:



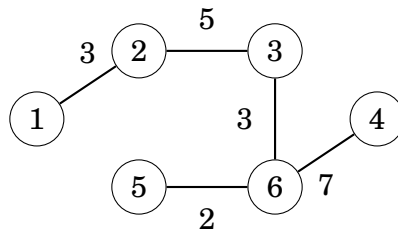
An arbitrary node can be the starting node, so let us choose node 1. First, we add node 2 that is connected by an edge of weight 3:



After this, there are two edges with weight 5, so we can add either node 3 or node 5 to the tree. Let us add node 3 first:



The process continues until all nodes have been included in the tree:



Implementation

Like Dijkstra's algorithm, Prim's algorithm can be efficiently implemented using a priority queue. The priority queue should contain all nodes that can be connected to the current component using a single edge, in increasing order of the weights of the corresponding edges.

The time complexity of Prim's algorithm is $O(n + m \log m)$ that equals the time complexity of Dijkstra's algorithm. In practice, Prim's and Kruskal's algorithms are both efficient, and the choice of the algorithm is a matter of taste. Still, most competitive programmers use Kruskal's algorithm.

第 16 章

有向グラフ - Directed graphs

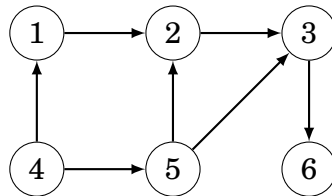
In this chapter, we focus on two classes of directed graphs: ここでは 2 つの特徴を持つグラフについて検討していきます。

- **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:** 各のノードの出次が 1 であるグラフ。このため、ただ一つの子を持つ。

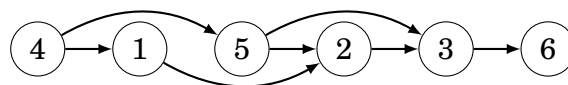
どちらの場合も、グラフの特別な性質に基づく効率的なアルゴリズムがあります。

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サイクル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。そこで、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することが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)

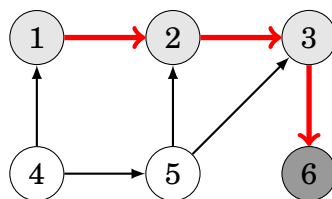
初期状態では、各ノードの状態は 0 です。初めて検索が到達すると、そのノードの状態は 1 になります。そして、そのノードのすべての探索が処理された後、そのノードの状態は 2 にします。

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わかります。

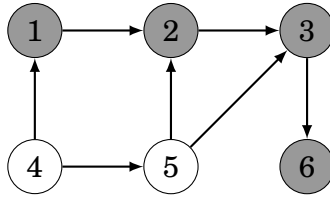
グラフがサイクルを含まない場合はこの条件以外の時で、全てのノードの状態が 2 になったときに各ノードの探索順をリストとすることで、トポロジカルソートとなります。

Example 1

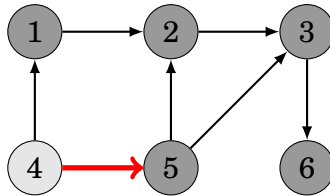
このグラフの例では 1,2,3,6 と探索が進みます。



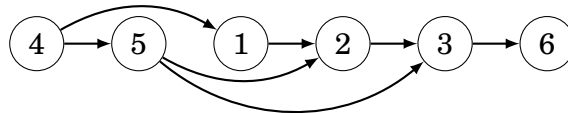
ここで、6 には探索する先がないので、状態 2 となり、リストに値が登録されます。次に、3,2,1 と追加されていきます。



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。



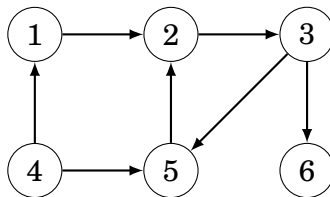
このようにリスト [6,3,2,1,5,4] を得ることができます。この逆順がトポロジカルソート [4,5,1,2,3,6] です。



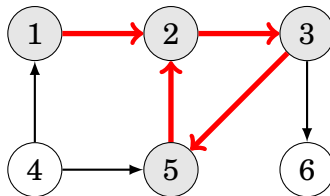
なお、トポロジカルソートは1つとは限りません。閉路を含まないグラフでは複数のトポロジカルソートが存在します。

Example 2

では閉路を含むのでトポロジカルソートできない場合を考えてみます。



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。

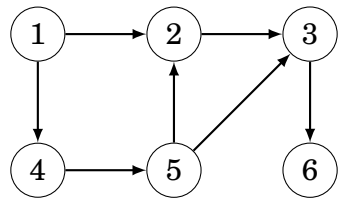
16.2 動的計画法 - Dynamic programming

有向グラフがサイクルを含まないのであれば動的計画法を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。

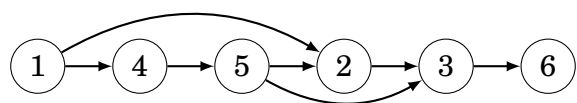
- 1 → 2 → 3 → 6
- 1 → 4 → 5 → 2 → 3 → 6
- 1 → 4 → 5 → 3 → 6

ノード 1 からノード x への経路の数を $\text{paths}(x)$ とすると、基本ケースとして $\text{paths}(1)=1$ である。そして、 $\text{paths}(x)$ の他の値を計算するために、再帰的に

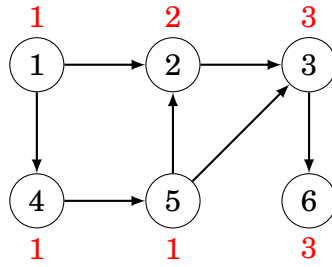
ノード 1 からあるノード x へのパスの経路数を $\text{paths}(x)$ と表現するとします。 $\text{paths}(1) = 1$ です。 $\text{paths}(x)$ は以下のように再帰的に求めています。

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \dots + \text{paths}(a_k)$$

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。

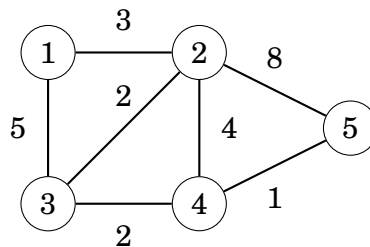


例えば、 $\text{paths}(3)$ の値を計算する場合、ノード 2 とノード 5 からノード 3 への辺があるので、 $\text{paths}(2) + \text{paths}(5)$ という式を用いることができる。 $\text{paths}(2) = 2$ 、 $\text{paths}(5) = 1$ なので、 $\text{paths}(3) = 3$ と結論付けられる。

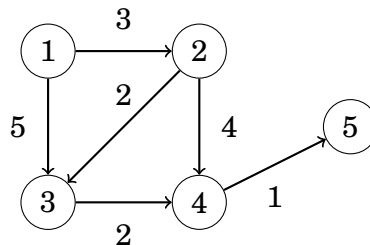
$\text{paths}(3)$ を求める時、ノード 2 と 5 からノード 3 への辺があるので、 $\text{paths}(2) + \text{paths}(5)$ を計算します。 $\text{paths}(2) = 2$ 、 $\text{paths}(5) = 1$ ですから、 $\text{paths}(3) = 3$ となります。

ダイクストラのアルゴリズムの拡張

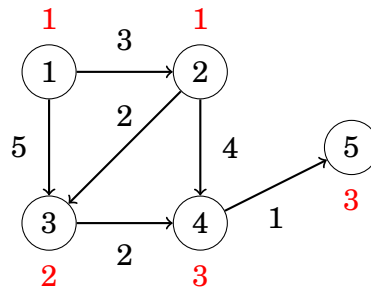
ダイクストラのアルゴリズムを利用すると、開始ノードから各ノードへの最短経路を使用してそのノードに到達する可能性を示す有向の閉路が存在しないグラフを得ることができます。このグラフに対しては動的計画法を適用することができます。以下を考えます。



ノード 1 からの最短経路は以下のようになります。



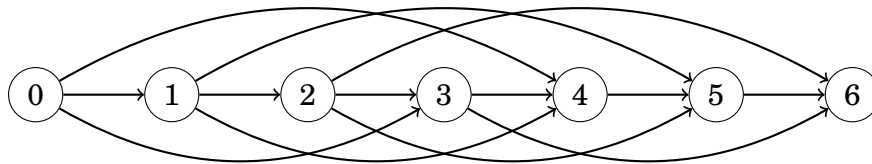
これでノード 1 から 5 までの最短経路を以下のよう求めることができます。



グラフによる問題表現 - Representing problems as graphs

動的計画問題の問題というのは有向無サイクル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。

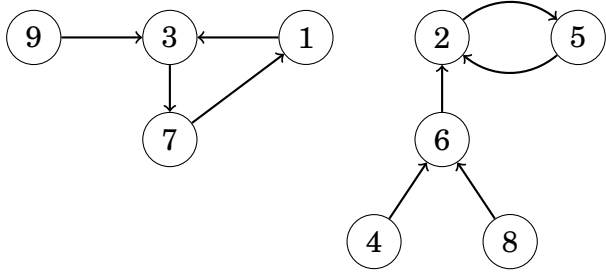
16.3 サクセスパス - Successor paths

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からなります。それぞれのコンポーネントには 1 つのサイクル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る。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。

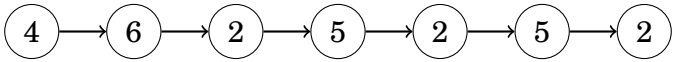
| x | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------------------|---|---|---|---|---|---|---|---|---|
| $\text{succ}(x)$ | 3 | 5 | 7 | 6 | 2 | 2 | 1 | 6 | 3 |

これは次のようなパスと考えられます。



後継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 歩歩くとノード 2 に到達することを表します。

Since each node of a successor graph has a unique successor, we can also define a function $\text{succ}(x,k)$ that gives the node that we will reach if we begin at node x and walk k steps forward. For example, in the above graph $\text{succ}(4,6) = 2$, because we will reach node 2 by walking 6 steps from node 4:



$\text{succ}(x,k)$ 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起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、

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

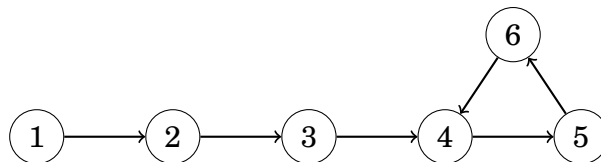
これは次のように展開されます。

$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

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。

16.4 閉路検出 - Cycle detection

閉路で終わる有向グラフを考えます。スタートから歩き始めたときに閉路の最初のノードは何で、サイクルはいくつのノードを含むか？を考えます。



今、ノード 1 から歩き始めたとき、閉路に属する最初のノードは 4 で、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、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 $O(n)$ の時間計算量であり、空間計算量です。

ここで閉路検出のためのアルゴリズムを紹介します。 $O(1)$ の空間計算量で良いアルゴリズム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 ^{*1}

2つのポインタ a 、 b を用いてグラフを探索していきます。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) 最初に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 がカナ猿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);
}
first = a;
```

そして、サイクル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++;
}
```

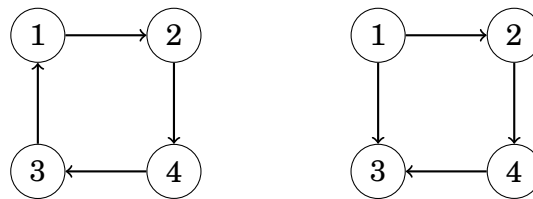
^{*1} The idea of the algorithm is mentioned in [46] and attributed to R. W. Floyd; however, it is not known if Floyd actually discovered the algorithm.

第 17 章

強連結 - Strong connectivity

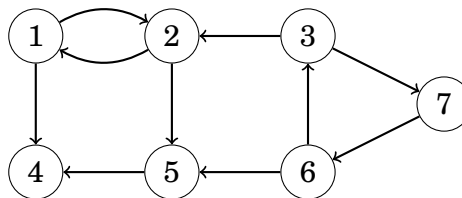
有向グラフにおいて、辺は一方方向にしか進めないで、あるグラフ上のあるノードから別のノードへの経路が存在することは保証されません。そのため、ノードとノードに辺があるとは違う新しい概念の定義に意味があります。

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** となります。例えば、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。

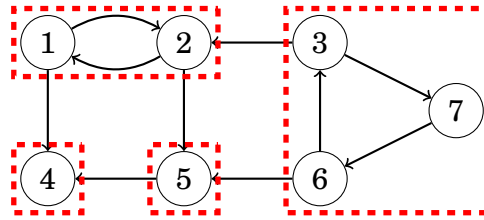


右のグラフは 2 から 1 へ到達できないので強連結グラフとはいえません。

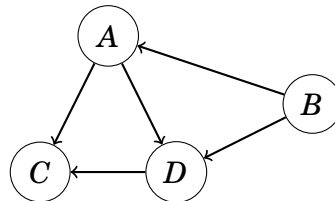
強連結成分 - strongly connected components は、グラフをできるだけ大きな強連結な部分に分解します。強連結成分は、元のグラフの閉路を含まない成分グラフを形成します。



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。

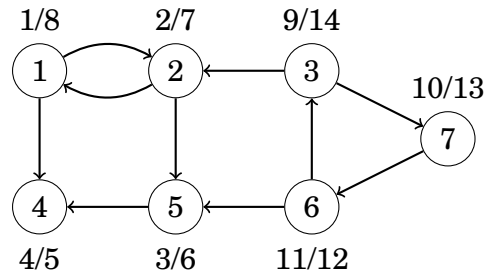
この成分グラフは閉路を含まない有向グラフであるため、元のグラフよりも処理しやすくなります。閉路が含まれないので、常にトポロジカルソートを構築し、16章で紹介したような動的計画法を利用することができる。

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。



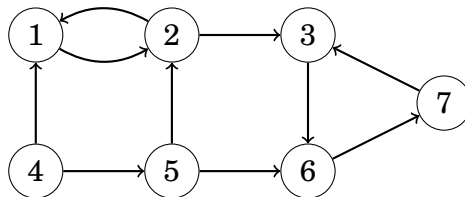
^{*1} According to [1], S. R. Kosaraju invented this algorithm in 1978 but did not publish it. In 1981, the same algorithm was rediscovered and published by M. Sharir [57].

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。

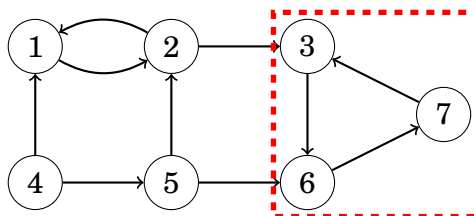
| node | processing time |
|------|-----------------|
| 4 | 5 |
| 5 | 6 |
| 2 | 7 |
| 1 | 8 |
| 6 | 12 |
| 7 | 13 |
| 3 | 14 |

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。

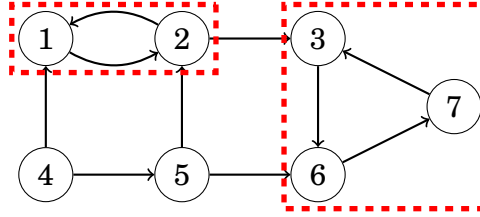


この後、最初の検索で作成されたノードのリストを逆順に捜査します。ノードが成分に属さない場合、新しい成分を作成してその探索中に見つかったすべての新しいノードを新しい成分に追加する深さ優先探索をおこなって行きます。このグラフの例では、最初のコンポーネントはノード 3 から始まっています (最初の探索で最後のノードは 3 だったことを思い出してください)。

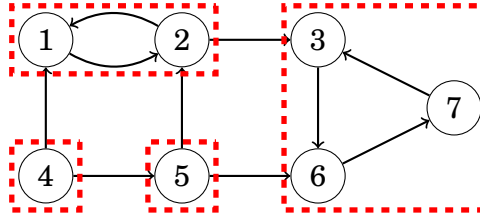


ここで、全てのグラフは反転していることで、成分が他の成分にリークすることはありません。

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ので、ノード1から次の成分の探索を開始します。



同じように次の成分として5と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 " \wedge " と " \vee " 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ことです。

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

という問題を考えると例えば次の割り当て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。

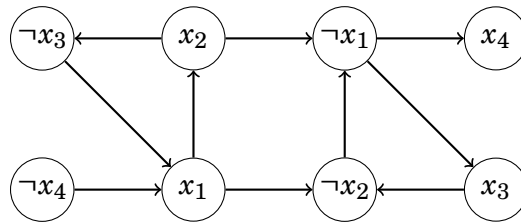
$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

つまり、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。

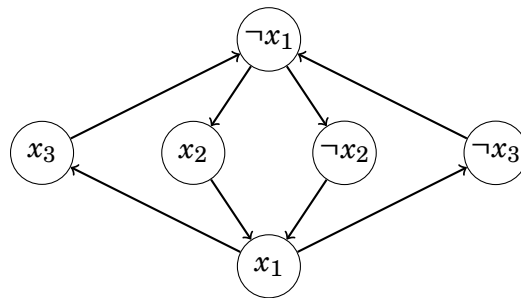
x_1 が false なら x_2 と $\neg x_2$ が true でないといけませんがこれは不可能です。 x_1 が true なら x_2 と $\neg x_2$ が false でないといけませんがこれも不可能です。

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 $\neg x_i$ があり、エッジは変数間の接続だとします。このとき、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 $\neg b_i \rightarrow a_i$ 。これは、 a_i が成立しない場合、 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。

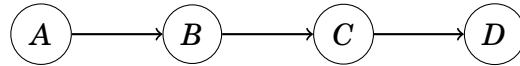


グラフを用いて式が真となるように変数の値を割り当てることが可能かどうかを調べられます。ある強連結成分に論理変数 x_i とその否定 $\neg x_i$ が同時に所属しなければ、この式を成立させる組み合わせが存在するといえます。逆に両ノードが同じ強連結成分に所属する時、その 2 つの変数は同時に満たされなければならない、成立

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 のグラフは以下のようなものでした。



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 3SAT 問題がある。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

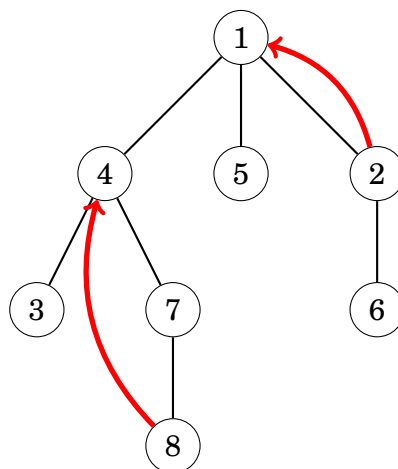
木に対するクエリ - Tree queries

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)。たとえば、次の木では、 $ancestor(2, 1) = 1$ and $ancestor(8, 2) = 4$ 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。

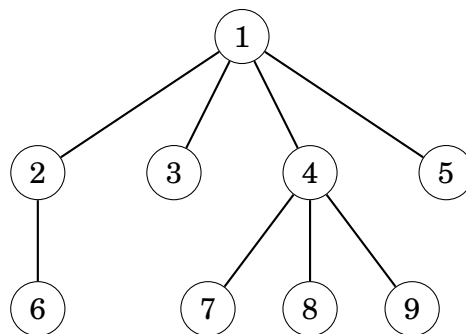
ここで 16.3 章で用いたのと同様の手法を用いれば、前処理を行うことで $ancestor(x, k)$ を $O(\log k)$ で効率的に求められます。 $k \leq n$ となる 2 の累乗について $ancestor(x, k)$ を事前計算します。例えば、上記の木に対する値は以下の通りになります。

| 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 |
| ... | | | | | | | | |

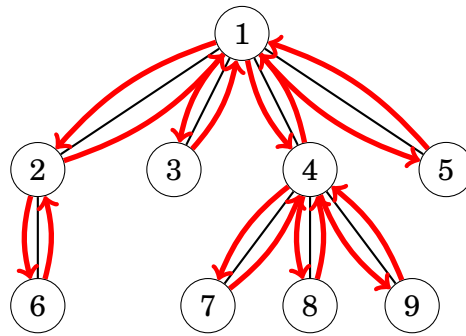
この前処理には各ノードに対して $O(\log n)$ の計算が必要なので $O(n \log n)$ の時間がかかります。この後、 $ancestor(x, k)$ の任意の値は、 k を各項が 2 のべき乗である和として表現することで $O(\log k)$ で計算可能となります。

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

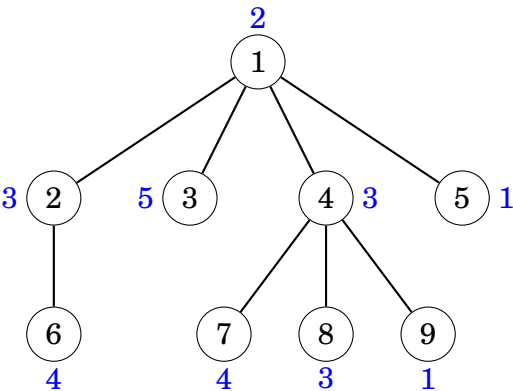
木における部分木は、木の探索順のある部分配列に対応し、その部分配列の最初の要素がその部分木の根となるようにします。例えば、以下の部分配列は、ノード 4 の部分木となります。

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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。例として、各ノードに値があるとします。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 |

この配列によって、まずサブツリーの大きさを求め、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 |

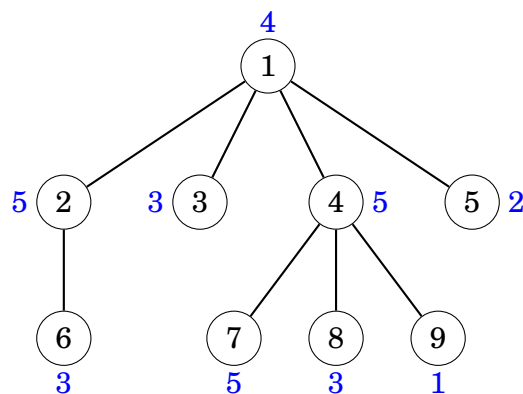
クエリに効率的に答えるにはこれを格納する構造体を工夫することが必要です。バイナリインデックスまたはセグメントツリーなどが適切でしょう。これらを用いると値の更新と値の総和の計算の両方を $O(\log n)$ で行うことができます。

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。

We can solve this problem like before, but now each value in the last row of the array is the sum of values on a path from the root to the node. For example, the

following array corresponds to the above tree:

| | | | | | | | | | |
|--------------|---|---|----|---|---|----|----|----|---|
| node id | 1 | 2 | 6 | 3 | 4 | 7 | 8 | 9 | 5 |
| subtree size | 9 | 2 | 1 | 1 | 4 | 1 | 1 | 1 | 1 |
| path sum | 4 | 9 | 12 | 7 | 9 | 14 | 12 | 10 | 6 |

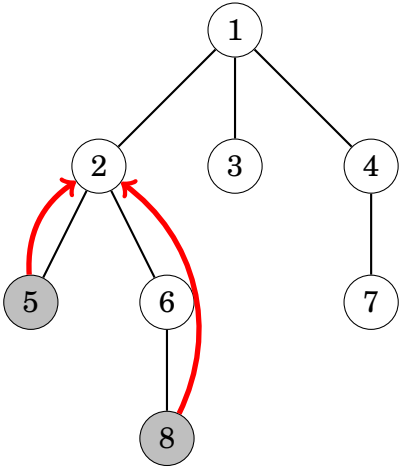
あるノードの値が x 増加した時、そのサブツリーの値は全て x 増加します。例えば、ノード 4 が 1 増加した時、以下のように変化が起こります。

| | | | | | | | | | |
|--------------|---|---|----|---|----|----|----|----|---|
| node id | 1 | 2 | 6 | 3 | 4 | 7 | 8 | 9 | 5 |
| subtree size | 9 | 2 | 1 | 1 | 4 | 1 | 1 | 1 | 1 |
| path sum | 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

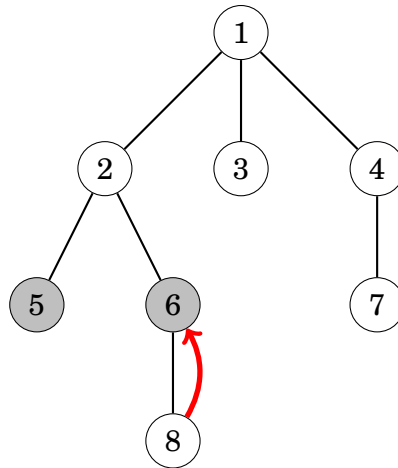
根付き木の 2 つのノードの**最小共通祖先 (LCA) - lowest common ancestor** とは、あるノードの部分木が 2 つのノードを含むような最も下のノードのことです。最も典型的な問題は 2 つのノードのペアが与えられるのでその問い合わせを効率的に行う問題です。例えば、以下の木では、ノード 5 と 8 の LCA はノード 2 です。



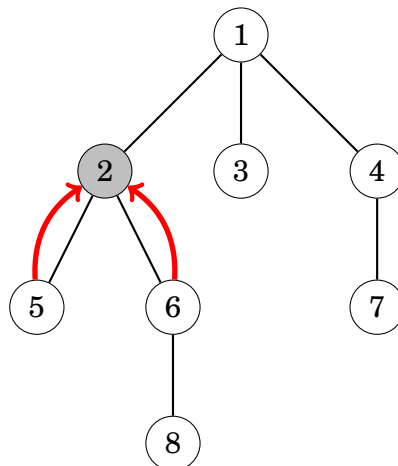
さて、LCA を求めるための 2 つの方法を紹介します。

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。



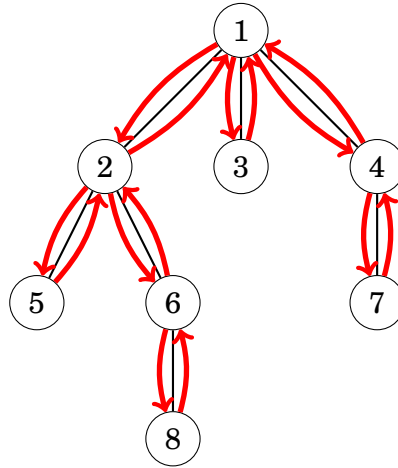
次に、両者のポインタを上方に移動させ、同じノードを指すようにするために必要な最小のステップ数を求めます。この後、ポインタが指すノードが LCA となります。この例では、両方のポインタを一段階上のノード 2(LCA) に移動させることになります。



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 |
|---------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| node id | 1 | 2 | 5 | 2 | 6 | 8 | 6 | 2 | 1 | 3 | 1 | 4 | 7 | 4 | 1 |
| depth | 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 |
|---------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| node id | 1 | 2 | 5 | 2 | 6 | 8 | 6 | 2 | 1 | 3 | 1 | 4 | 7 | 4 | 1 |
| depth | 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].

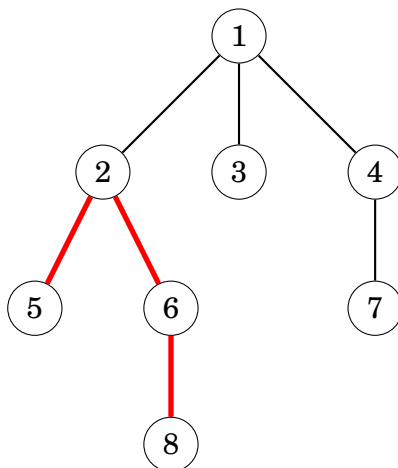
理すればよい。配列は静的なので、 $O(n \log n)$ の前処理をした後、 $O(1)$ 時間でこのようなクエリを処理することができます。(訳注: Sparse Table などを用いることでこれが実現できます)

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、木を根付きとして考えます。この後、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 と 8 の最小公倍数の祖先はノード 2 である。ノードの深さは $\text{depth}(5) = 3$, $\text{depth}(8) = 4$, $\text{depth}(2) = 2$ なので、ノード 5 と 8 の間の距離は $3 + 4 - 2 \cdot 2 = 3$ である。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。

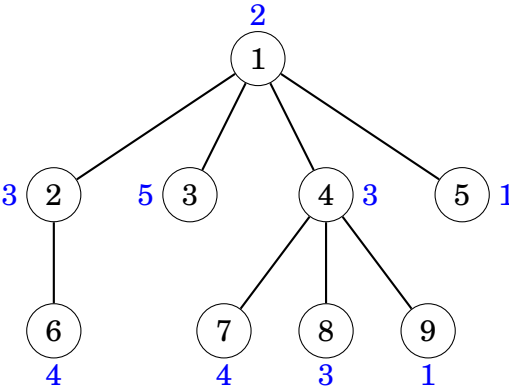
オンラインアルゴリズムは、次のクエリがくる前にそのクエリに応答するように、クエリを順番に処理することができます。これらが必要ないオフライン - *offline* なアルゴリズムに注目します。オフラインであるとは、任意の順序で回答可能な問い合わせのセットが与えられます。つまり、最初から全てのクエリが分かっている

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。

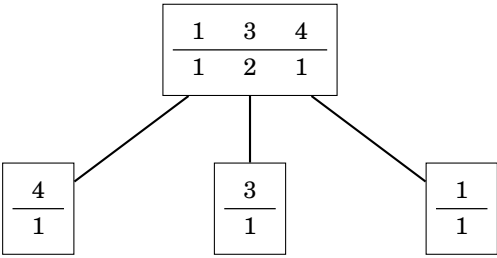
データ構造のマージ - Merging data structures

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い合わ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。



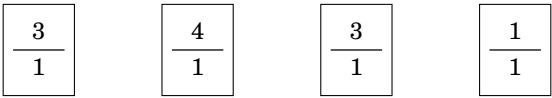
この問題では、map 構造を用いてクエリに答えることができます。例えば、ノード 4 とその部分木のノードの map は以下の通りになります。



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 の子を訪問して 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なります。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。 s の子を訪問し、各 u で $d[s]$ と $d[u]$ をマージする。 $d[u]$ 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。

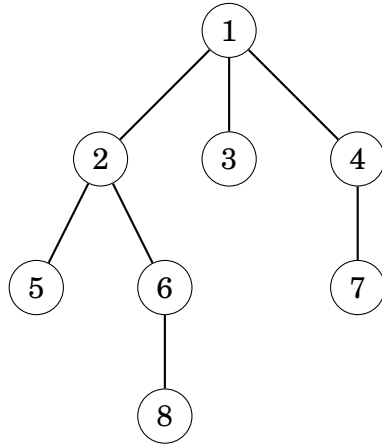
このアルゴリズムは、入力としてノードのペアのセットが与えられ両ノードの最小公倍数の祖先を決定します。まず、深さ優先の木探索によりノードの不連続な集合を保持します。初期状態では、各ノードは別々の集合に属している。また、各集合に対して、その集合に属する木における最も深さが高い（根に近い）ノードを保存しておきます。

TODO: この文章確認。 x, y に対して y ?

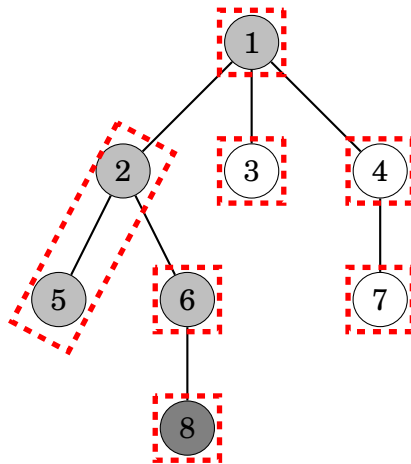
When the algorithm visits a node x , it goes through all nodes y such that the lowest common ancestor of x and y has to be found. If y has already been visited, the algorithm reports that the lowest common ancestor of x and y is the highest node in the set of y . Then, after processing node x , the algorithm joins the sets of x and its parent.

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。

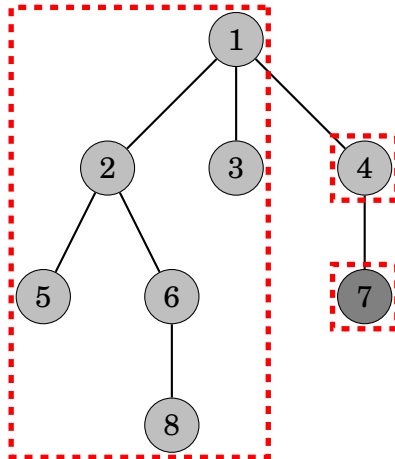
^{*2} This algorithm was published by R. E. Tarjan in 1979 [65].



次の図で灰色のノードは既に訪問したノードを表し、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。アルゴリズムがノード 8 を訪問します。この時にノード 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

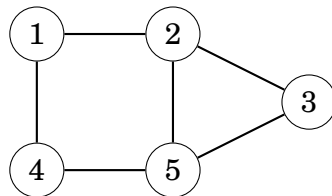
この章では次の 2 つの経路を扱います。

- An **オイラー経路 - Eulerian path** とは、各辺をちょうど 1 回ずつ通る経路のことです。
- A **ハミルトン経路 - Hamiltonian path** 各ノードをちょうど 1 回ずつ訪れる経路のことです

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。

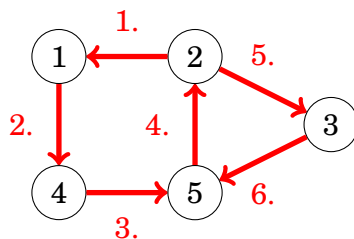
19.1 オイラー経路 - Eulerian paths

オイラー経路 - 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。

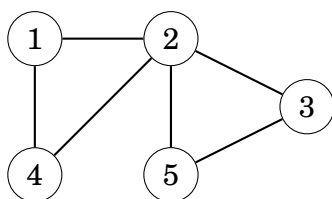


このグラフはノード 2 からノード 5 までのオイラー的な経路を持ちます。

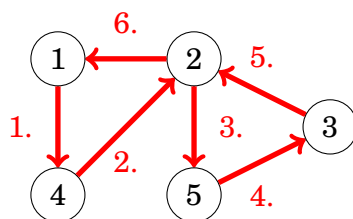
^{*1} L. Euler studied such paths in 1736 when he solved the famous Königsberg bridge problem. This was the birth of graph theory.



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、

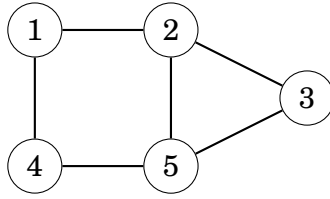
- 全てのノードの次数が偶数である
- ちょうど 2 つのノードだけの次数が奇数で、他のすべてのノードの次数が偶数である

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、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。

For example, in the graph



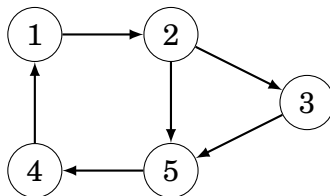
ノード 1、3、4 は次数 2 でノード 2、5 が次数 3 で、ちょうど 2 つのノードが奇数の次数を持ちます。ノード 2 と 5 の間にはオイラー経路が存在するが、このグラフにはオイラー閉路は存在しません。

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、

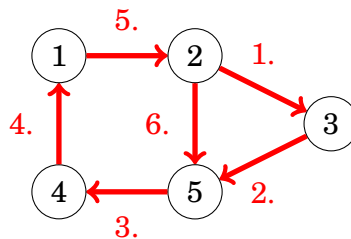
- 全てのノードにおいて出次数と入次数が等しい あるいは、
- ある一つのノードで入次数が出次数より 1 だけ大きく、他の一つのノードで出次数が入次数より 1 だけ大きく、他の全てのノードでは出次数と入次数が等しい

という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、終了が入次であるオイラー閉路が含まれます。

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。

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。そして、すべての辺が閉路に追加されるまで繰り返します。

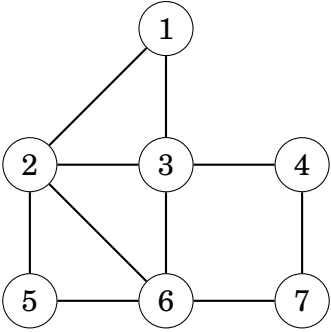
このアルゴリズムは、閉路に属しているノードで閉路に含まれていない出次辺を持つノード x を処理して閉路を拡張していく。このノード x から、まだ閉路に含まれていないエッジのみを含む新しい経路を構築する。するとそのパスはノード x に戻り、サブ閉路が作られる。

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、2つの奇数次ノードの間に余分なエッジを追加します

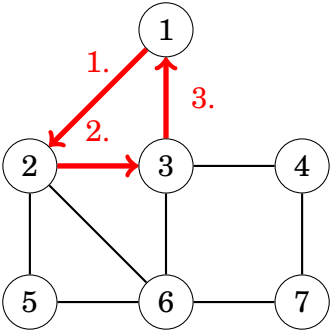
実際にヒールホルツァーのアルゴリズムが無向グラフに対してどのようにオイラー路を構築するかを見ていきましょう。

^{*2} The algorithm was published in 1873 after Hierholzer's death [35].

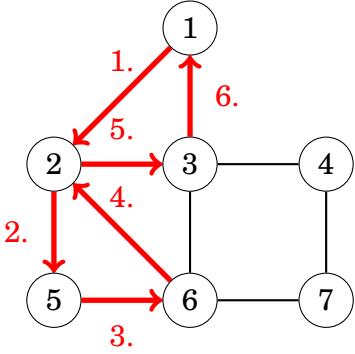
Example



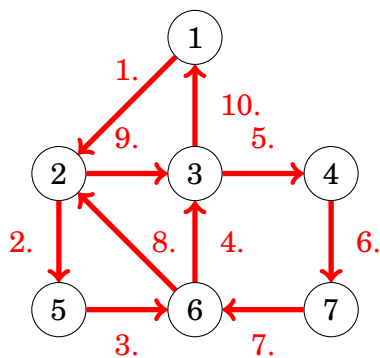
このアルゴリズムでは、まずノード 1 から始まる回路を作成するとします。たとえば $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ です。



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。



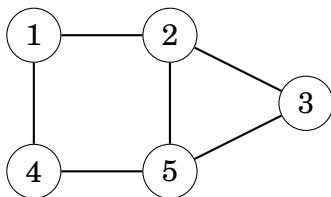
そして、 $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ を追加します。



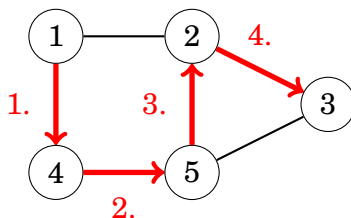
これによってオイラー閉路が完成しました。

19.2 ハミルトン路 - Hamiltonian paths

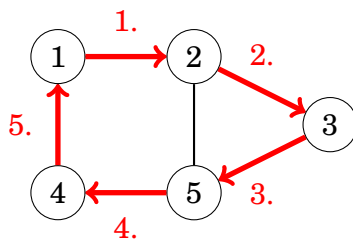
ハミルトン路 - Hamiltonian path というのは各ノードをちょうど1回ずつ訪問するパスのことです。



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。上のグラフも、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、グラフはハミルトン路を含む。

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、ハミルトンパス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也没有ありません。もし、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ね。

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

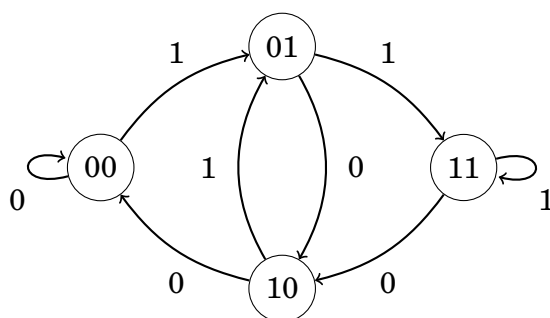
デ・ブルーイエン配列 - De Bruijn sequence は、文字数 k の固定アルファベットの文字列に対して、長さ n のすべての文字列を部分文字列としてちょうど一度だけ含む文字列のことです。このような文字列の長さは、 $k^n + n - 1$ 文字です。例

例えば、 $n = 3, k = 2$ のとき、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 and 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 文字があるので、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、盤上のマスをノードと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。この探索は、完全なツアーを素早く見つけるために、ヒューリスティックを用いることでより効率的に行うことができます。

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。

例えば、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。この選択の後には、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になります。

^{*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.

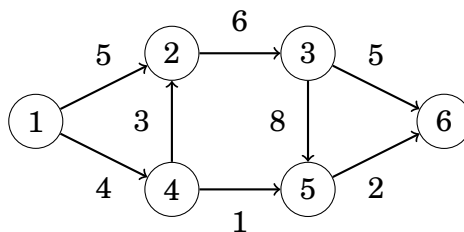
第 20 章

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。

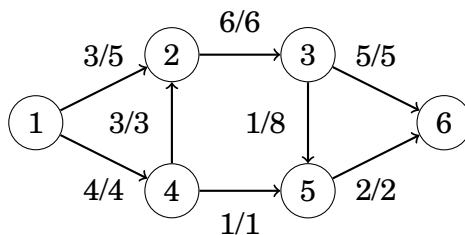
- **最大流の発見 - Finding a maximum flow:** あるノードから他のノードに送ることができるフローの最大の量は？
- **最小カットの発見 - Finding a minimum cut:** グラフを分割する辺のカットを最小にするには？

共通する前提は、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です。次のグラフではノード 1 がソース、ノード 6 がシンクといえます。



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。

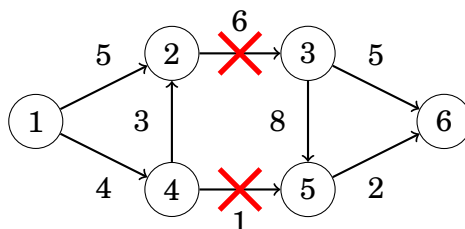


v/k という表記は、 v の流量のフローが k 単位の容量を持つエッジを経由していることを意味します。送信元が $3+4$ のフローを送信し、シンクが $5+2$ ユニットのフローを受信するので、このグラフのフローの大きさは 7 です。このグラフではシンクにつながるエッジの総容量は 7 なので、このフローが最大であることは容易に理解できるでしょう。

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。



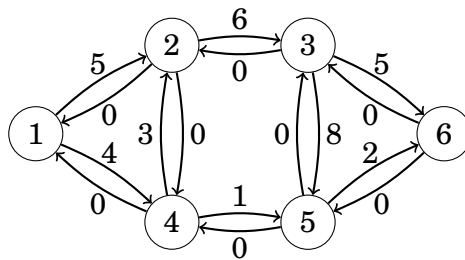
エッジを削除すると、ソースからシンクへのパスは存在しなくなります。削除されたエッジの重みは 6 と 1 であるため、このカットの大きさは 7 です。

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。空のフローから始まり、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。最後に、これ以上フローを増やせなくなったとき、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、グラフの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。

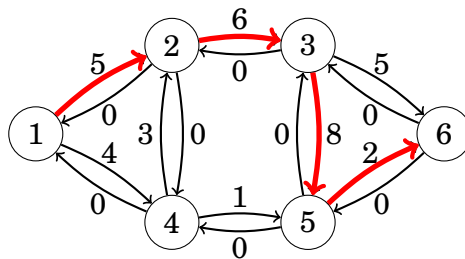
これを示したグラフは次のようになります。



アルゴリズムの説明 - Algorithm description

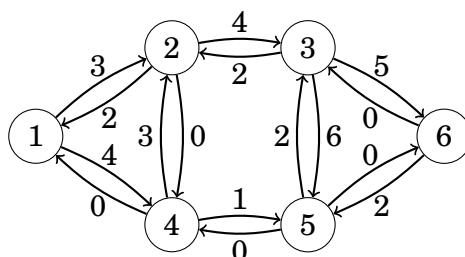
この処理は複数のラウンドで構成されます。各ラウンドで、アルゴリズムはソースからシンクへのパスを見つけ、そのパス上の各エッジが正の重みを持つようにします。複数の可能な経路がある場合は、そのうちのどれかを選択します。

例えば、次のようなパスを選択したとしましょう。



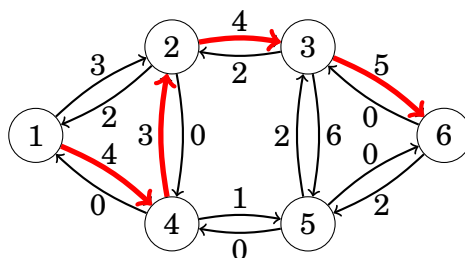
経路を選択した後、フローはその経路に流せる最大の量である x 単位で経路を増やします。ここで、 x は経路上の最小のエッジの重みとなります。また、経路上の各辺の重みを x だけ減少させ、各逆辺の重みを x だけ増加させます。上のパスで

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。



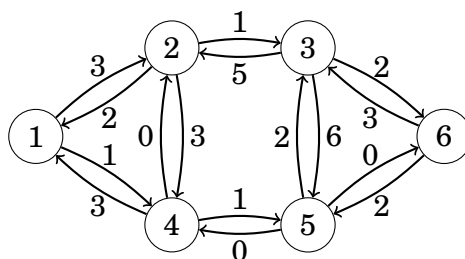
つまり、フローを増やすとそのエッジを通ることができるフローが減少するという操作です。一方、フローを別の方法で流した方が有益であることが判明した場合、グラフの逆エッジを使用してそのフローを後でキャンセル（減らす）ことができます。

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。

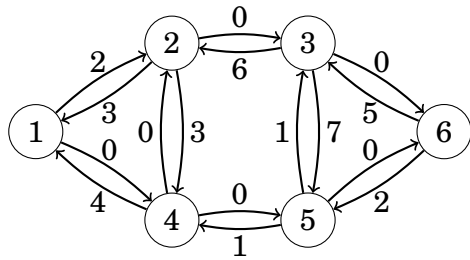


このパスの最小の重さは3なので、このパスによってフローは3増加し、パス処理後の総フローは5となります。

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。したがって、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る。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、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だけ増加させ、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。

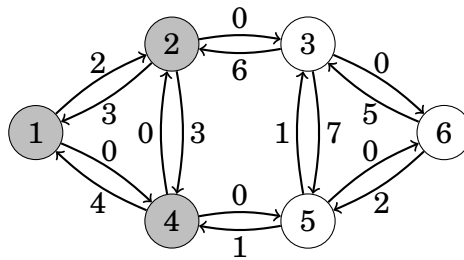
[2] は深さ優先探索により、選択する各辺の重みが少なくとも閾値以上である経路を見つけるものである。初期状態では、閾値はある大きな数、例えばグラフの全てのエッジ重みの和からスタートします。パスが見つからないときは閾値を2で割ります。このアルゴリズムの時間計算量は $O(m^2 \log c)$ であり、ここで c は初期の閾値です。深さ優先探索で経路を見つけることができるため、スケーリングアルゴリズムの方が実装が簡単です。

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。

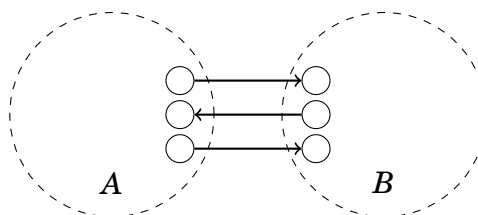
例のグラフでは、 A にはノード 1、2、4 が含まれる。



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 $6 + 1 = 7$ に相当する。

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よう。



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。

20.2 素なパス - Disjoint paths

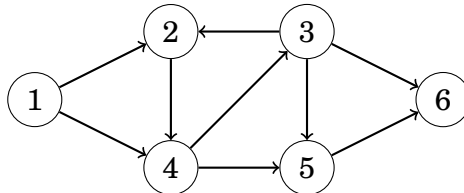
多くのグラフ問題は、最大流問題に還元することで解くことができます。1つの例をあげます。

ソースとシンクを持つ有向グラフが与えられ、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。

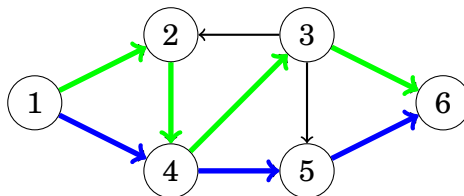
辺素なパス

まず、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することである。

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る。

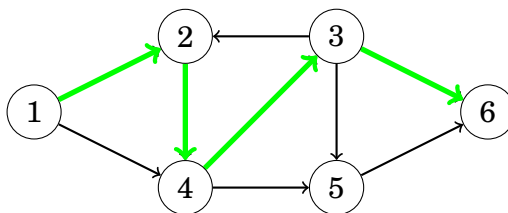


各エッジの容量を 1 とすると、辺素なパスの最大数はグラフの最大フローに等しいことが判明しました。最大流が構成された後の辺素なパスの経路は、ソースからシンクへの経路を貪欲にたどることで見つけることができます。

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。ここで、ソースとシンクは複数回登場しても良いです。

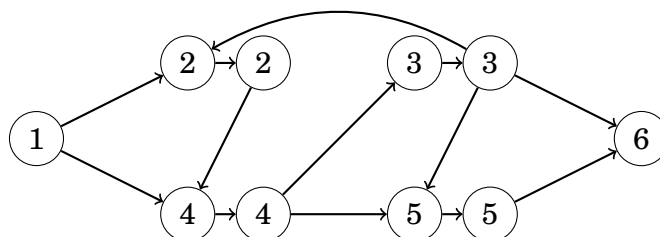
このグラフの場合は 1 となります。



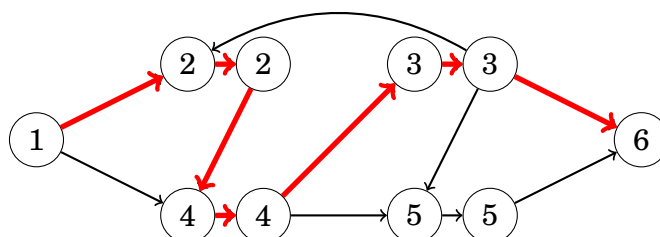
この問題も最大流問題とすることができます。各ノードは最大でも 1 つの経路にしか現れないので、ノードを通過する流れを制限しなければなりません。このための標準的な方法は、各ノードを 2 つのノードに分割し、最初のノードは元のノードの入次辺を持ち、2 番目のノードが出次辺を持つようにします。そして、最初のノードから 2 番目のノードに向かう新しいエッジが存在するようにします。

この例では、グラフは次のようになる。

In our example, the graph becomes as follows:



最大流は以下ようになります。



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。

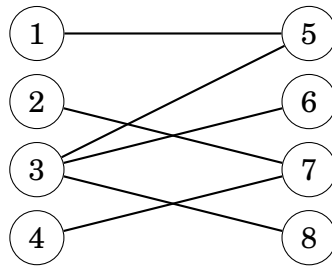
20.3 最大マッチング - Maximum matchings

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。

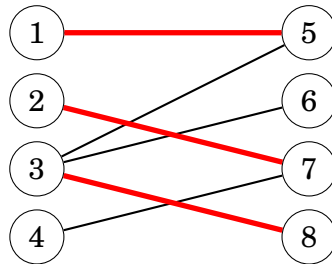
一般的なグラフでも最大マッチングを求める多項式アルゴリズムがあるが [17]、そのようなアルゴリズムは複雑で、プログラミングコンテストで見られることはほぼありません。しかし、二部グラフでは、最大マッチング問題は最大フロー問題に還元できます。

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。

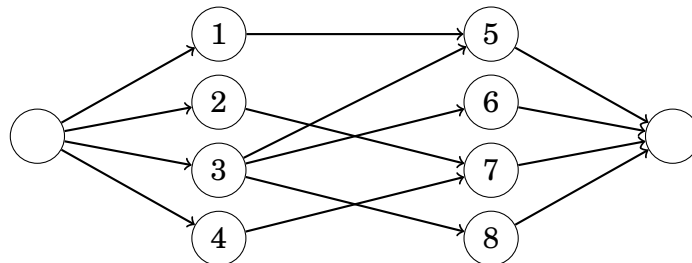


このグラフの最大マッチングの大きさは3です。

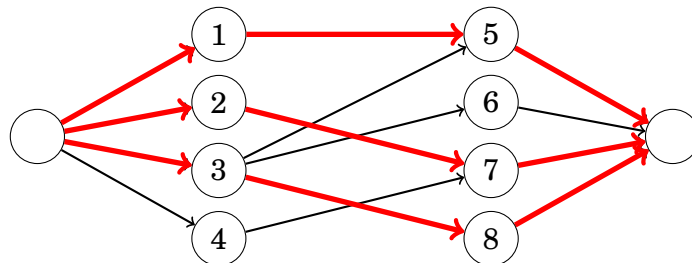


このグラフにソースとシンクという2つのノードを新たに加えることで、二部最大マッチング問題を最大フロー問題に還元することができます。ソースから各左ノードへ、各右ノードからシンクへのエッジを追加しましょう。この後、グラフ内の最大流の大きさは、元のグラフの最大マッチングの大きさと等しくなります。

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。



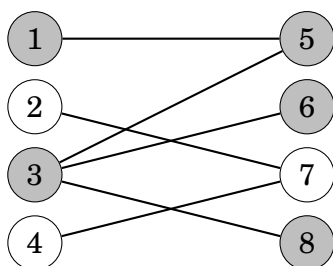
ホールの定理 - Hall's theorem

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

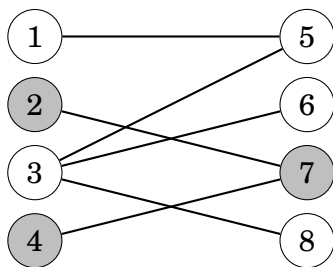
数が同じとき、グラフのすべてのノードを含む**完全マッチング**を構成できるかどうか、ホールの定理で判定できます。

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。



$|X| = 2$ and $|f(X)| = 3$ 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、 $|X| = 2$, $|f(X)| = 1$ なので、ホールの定理の条件は成立しません。つまり、グラフの完全マッチングを形成することはできません。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ます。

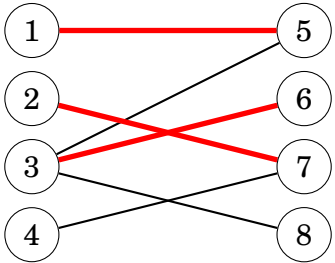
この場合、そのようなマッチングが成立しない理由を集合 X で説明しましょう。 X は $f(X)$ よりも多くのノードを含むので、 X のすべてのノードに対応するペアは存在しません。例えば、上のグラフでは、ノード 2 とノード 4 はともにノード 7 と接続されて欲しいですが、それは不可能です。

ケーニヒの定理 - Knig's theorem

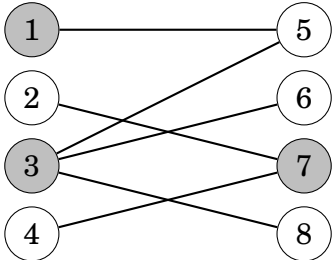
A **最小点被覆 - minimum node cover** は、グラフの各辺が少なくとも 1 つのエンドポイントを持つようなノードの最小集合です。一般的なグラフでは、点被覆の最小値を求めることは NP-Hard な問題である。しかし、グラフが二部グラフであ

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。

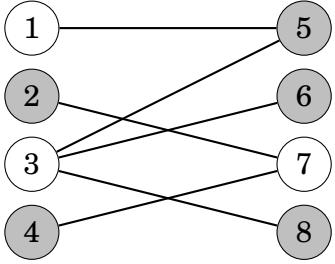
では、次のような最大マッチング 3 のグラフを考えます。



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。

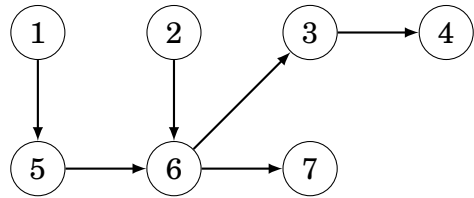


20.4 辺被覆 - Path covers

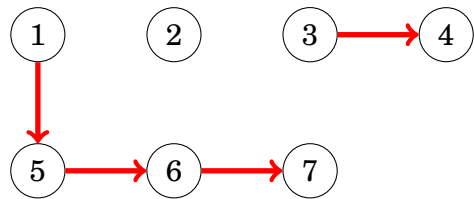
辺被覆 - path cover はグラフの各ノードが少なくとも 1 つのパスに属するようなグラフのパスの集合のことです。閉路を含まない有向グラフにおいて、最小辺被覆を求める問題は、別のグラフで最大流を求める問題に帰着できることがわかります。

点素な辺被覆 - Node-disjoint path cover

In a 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。例として、次のようなグラフを考えてみよう。



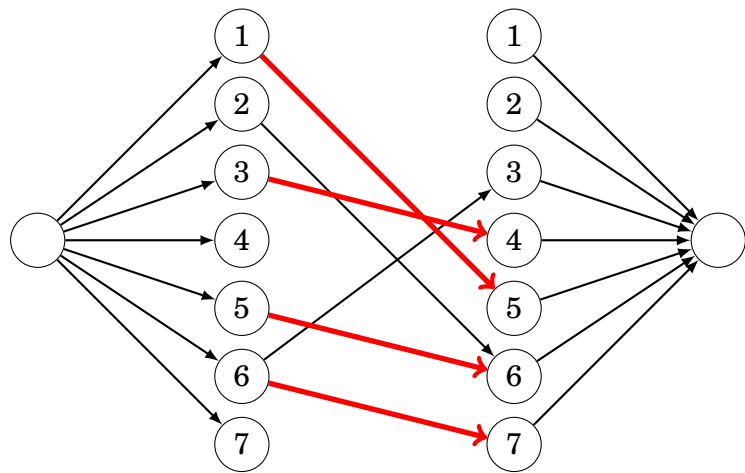
このグラフの最小の点素な辺被覆は、3つのパスから構成されます。



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する。

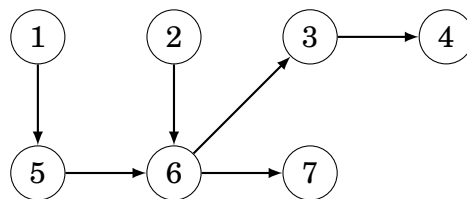
この結果得られたグラフの最大マッチングは、元のグラフの最小の点素な辺被覆に対応します。例えば、上記のグラフに対する以下のマッチンググラフは、サイズ 4 の最大マッチングを含んでいます。



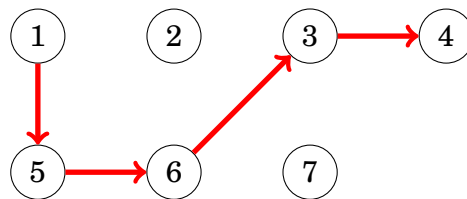
マッチンググラフの最大マッチングの各エッジは、元のグラフの最小の点素な辺被覆のエッジに対応します。したがって、最小の点素な辺被覆の最小サイズは $n - c$ といえます。ここで n は元グラフのノード数、 c は最大マッチングのサイズである。

一般的な辺被覆 - General path cover

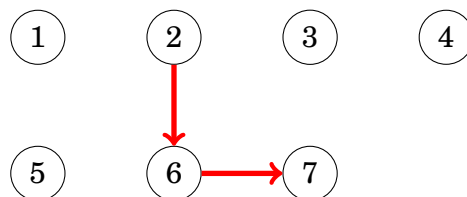
一般的な辺被覆 - **general path cover** は1つのノードが複数のパスに属することができる辺被覆のことです。1つのノードは複数回パスで使われることがあるため、最小の一般的な辺被覆は最小の点素な辺被覆より小さくなることがあります。先ほどのグラフを例にとります。



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

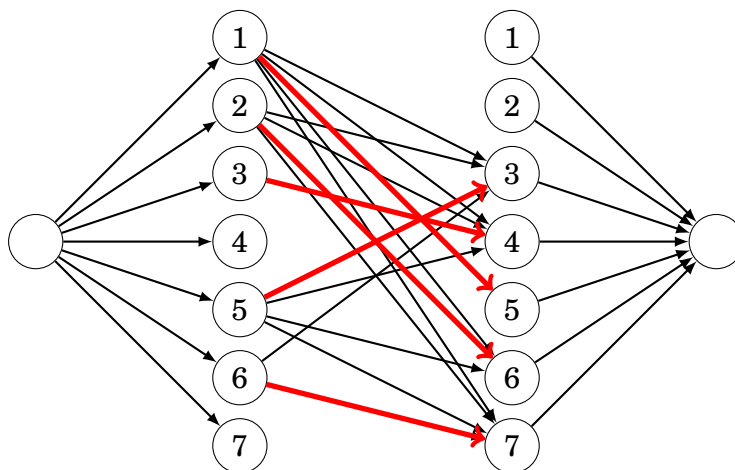


次に2つ目は、



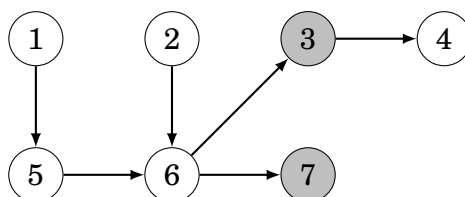
最小の一般的な辺被覆は、最小の点素な辺被覆とほぼ同様に求めることができます。マッチンググラフに新しい辺をいくつか追加し、元のグラフに a から b への経路があるときに必ず辺 $a \rightarrow b$ が存在するようにすればよいです (これはいくつかの辺を経由する場合もある)。

上記のグラフのマッチンググラフは次のようになる。



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、しかし興味深い分野です。

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 and 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**です。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、 $\alpha_1, \alpha_2, \dots, \alpha_k$ は正の数である。例えば、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 の素因数の個数を **The number of factors of a number n is**

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

とします。なぜなら、各素数 p_i に対して、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 . たとえば 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。つまり、 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 $P = \{p_1, p_2, \dots, p_n\}$ で構成することができます。すべての素数を含むことになる。例えば、 $p_1 = 2, p_2 = 3, p_3 = 5$ です。ですが、 P 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 and $p + 2$ は素数である。
- **ルジャンドル予想 TODO (Legendre's conjecture):** n^2 and $(n + 1)^2$ の間には必ず素数が存在する。なお、 n は生成数 where n is any positive integer.

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、 $a \leq \sqrt{n}$ か $b \leq \sqrt{n}$ であるから、確実に 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、 $24 = 2^3 \cdot 3$ 、ですから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**) 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** を構築する。`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、 $2 \dots n$ 順番にを 1 回ずつみていきます。新しい素数 x が見つかると、 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ている。

```
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
```

```

for (int u = 2*x; u <= n; u += x) {
    sieve[u] = x;
}
}

```

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。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。

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \cdots + n/n = O(n \log n).$$

In fact, the algorithm is more efficient, because the inner loop will be executed only if the number x is prime. It can be shown that the running time of the algorithm is only $O(n \log \log n)$, a complexity very near to $O(n)$.

ユークリッドの互除法 - Euclid's algorithm

a と b の最大公約数 (**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 (**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。

^{*1} Euclid was a Greek mathematician who lived in about 300 BC. This is perhaps the first known algorithm in history.

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}
```

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**), 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 $\varphi(n)$ で示され、1 から n の n と素な個数です。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、 $\varphi(12) = 2^1 \cdot (2 - 1) \cdot 3^0 \cdot (3 - 1) = 4$. ここで、 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

$x^n \bmod m$ を高速に求めることができます。以下のように $O(\log n)$ で求めることができます。

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ is even} \\ x^{n-1} \cdot x & n \text{ is odd} \end{cases}$$

ここで重要なのは、偶数 n の場合、 $x^{n/2}$ の値を一度だけ計算していることです。 n が偶数の場合、 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、

$$x^{m-1} \bmod m = 1$$

となる。また、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 の環境下で 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ます。

注意しなければいけないのは、 mod の逆数とは常に存在するわけではないことです。例えば、 $x = 2$ 、 $m = 4$ とすると、

$$xx^{-1} \bmod m = 1$$

であり、解けません。なぜなら、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、 mod の逆元が存在する時、以下がなりたちます。

$$x^{-1} = x^{\varphi(m)-1}.$$

If m is prime, the formula becomes もし、 m が素数であるなら、次のようにいえます。

$$x^{-1} = x^{m-2}.$$

例を示します。

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

mod 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きに、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 と c は定数で x と y を求めたいとします。Each number in the equation has to be an integer. 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は、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出来ます。

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

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、全ての解のペアは以下のように示します。

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、

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

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 つの二乗の和で示せるというもので、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) もまた、ピタゴラスの定理を満たす組です。これらの組みは a, b, c が共に素数であれば**原始項 (primitive)** であり、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る。

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

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つけられます。

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

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学問です。これらは通常、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。

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \cdots + f(n-1) & n > 0 \end{cases}$$

$f(0)=1$ は自明で、これは空集合が数 0 を表すからである。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。なお、この時、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て、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、1 つ目の部分集合には k 個の要素を含み、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。

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1 b^{n-1} + \binom{n}{n}a^0 b^n.$$

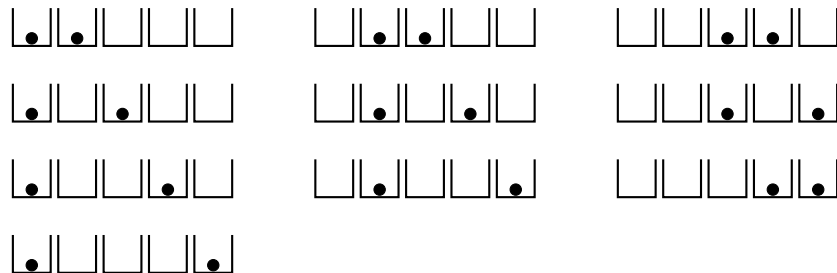
二項係数はパスカルの三角形 - **Pascal's triangle** と密接な関係があります。

$$\begin{array}{ccccccc} & & & & 1 & & & & \\ & & & & 1 & & 1 & & \\ & & & 1 & & 2 & & 1 & \\ & & 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & & \dots & & \dots & & \dots & & \dots \end{array}$$

箱とボールの問題 - Boxes and balls

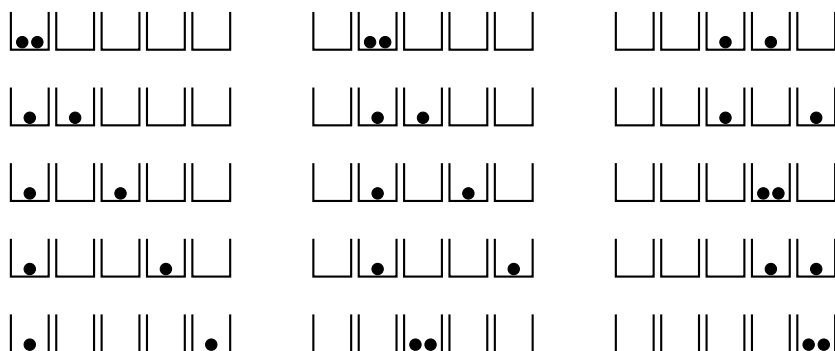
”箱とボールの問題 - Boxes and balls” は二項係数を考える上で非常に有名な問題です。 k 個の玉を n 個に入れる方法を数えます。3 つのシナリオを考えてみましょう。

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。



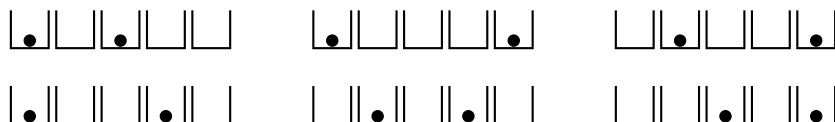
このシナリオでは答えは二項係数そのままです。 $\binom{n}{k}$ となります。

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、一番左の箱に立っているとする。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。しかし隣り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、隣り合う 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る。

- 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。ここで、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。

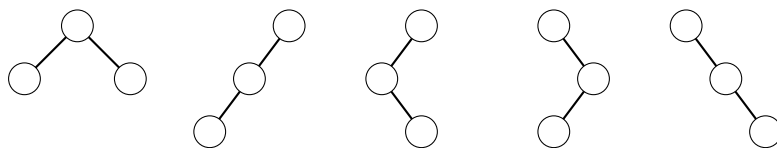
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

木の数え上げ - Counting trees

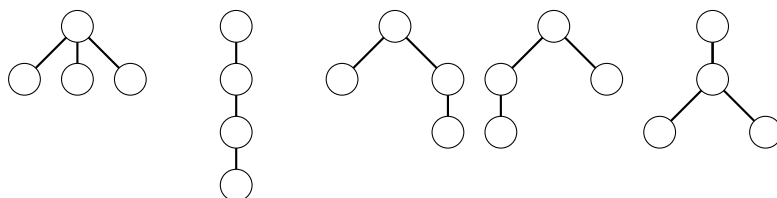
Catalan numbers are also related to trees: カタラン数は木とも関係があります。

- 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。

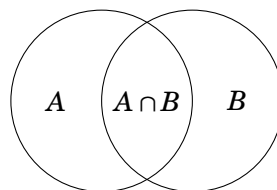


22.3 包除原理 - Inclusion-exclusion

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、

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

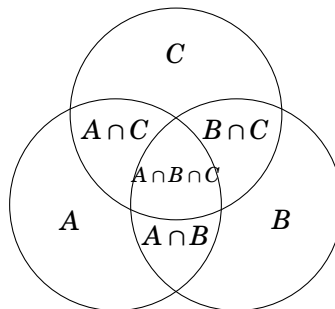
となり、 A と B は集合であり、 $|X|$ という表現は X の集合の大きさです。これを図に示します。



$A \cup B$ の大きさを計算するには、少なくとも 1 つの円に属する領域の面積に対応する組合わせを計算することです。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、偶数の集合を含む場合は減算します。

また、重なる部分のサイズを計算する式は以下のようになります。

$$|A \cap B| = |A| + |B| - |A \cup B|$$

集合が 3 つの場合は次のようになります。

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Derangements

TODO: 全体的に。 As an example, let us count the number of **derangements** of elements $\{1, 2, \dots, n\}$, i.e., permutations where no element remains in its original place. For example, when $n = 3$, there are two derangements: $(2, 3, 1)$ and $(3, 1, 2)$.

One approach for solving the problem is to use inclusion-exclusion. Let X_k be the set of permutations that contain the element k at position k . For example, when $n = 3$, the sets are as follows:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

Using these sets, the number of derangements equals

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

so it suffices to calculate the size of the union. Using inclusion-exclusion, this reduces to calculating sizes of intersections which can be done efficiently. For example, when $n = 3$, the size of $|X_1 \cup X_2 \cup X_3|$ is

$$\begin{aligned} & |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

so the number of solutions is $3! - 4 = 2$.

It turns out that the problem can also be solved without using inclusion-exclusion. Let $f(n)$ denote the number of derangements for $\{1, 2, \dots, n\}$. We can use the following recursive formula:

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

The formula can be derived by considering the possibilities how the element 1 changes in the derangement. There are $n - 1$ ways to choose an element x that replaces the element 1. In each such choice, there are two options:

Option 1: We also replace the element x with the element 1. After this, the remaining task is to construct a derangement of $n - 2$ elements.

Option 2: We replace the element x with some other element than 1. Now we have to construct a derangement of $n - 1$ element, because we cannot replace the element x with the element 1, and all other elements must be changed.

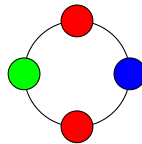
22.4 Burnside's lemma

Burnside's lemma can be used to count the number of combinations so that only one representative is counted for each group of symmetric combinations. Burnside's lemma states that the number of combinations is

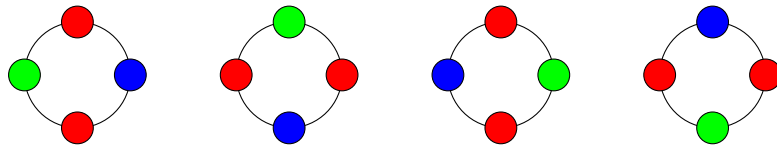
$$\sum_{k=1}^n \frac{c(k)}{n},$$

where there are n ways to change the position of a combination, and there are $c(k)$ combinations that remain unchanged when the k th way is applied.

As an example, let us calculate the number of necklaces of n pearls, where each pearl has m possible colors. Two necklaces are symmetric if they are similar after rotating them. For example, the necklace



has the following symmetric necklaces:



There are n ways to change the position of a necklace, because we can rotate it $0, 1, \dots, n-1$ steps clockwise. If the number of steps is 0, all m^n necklaces remain the same, and if the number of steps is 1, only the m necklaces where each pearl has the same color remain the same.

More generally, when the number of steps is k , a total of

$$m^{\gcd(k,n)}$$

necklaces remain the same, where $\gcd(k, n)$ is the greatest common divisor of k and n . The reason for this is that blocks of pearls of size $\gcd(k, n)$ will replace each other. Thus, according to Burnside's lemma, the number of necklaces is

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

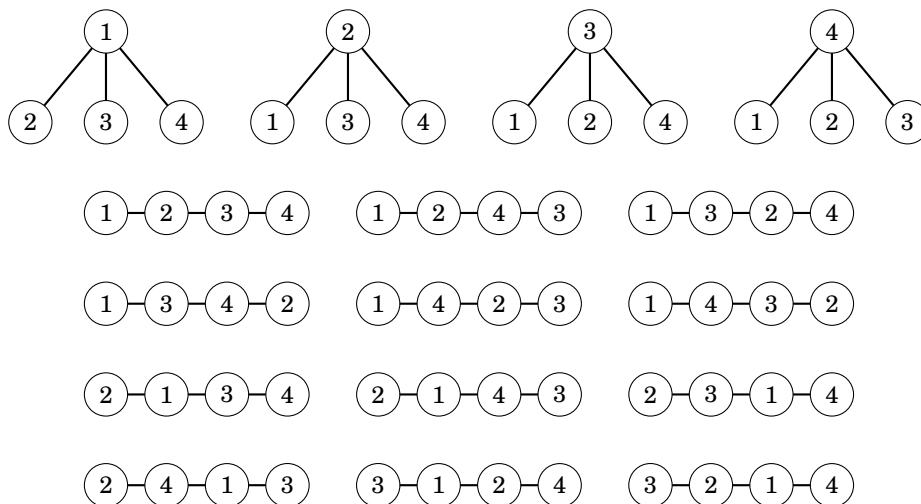
For example, the number of necklaces of length 4 with 3 colors is

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 Cayley's formula

Cayley's formula states that there are n^{n-2} labeled trees that contain n nodes. The nodes are labeled $1, 2, \dots, n$, and two trees are different if either their structure or labeling is different.

For example, when $n = 4$, the number of labeled trees is $4^{4-2} = 16$:

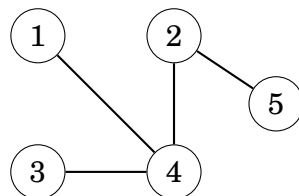


Next we will see how Cayley's formula can be derived using Prüfer codes.

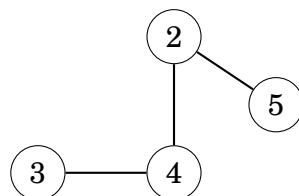
Prüfer code

A **Prüfer code** is a sequence of $n - 2$ numbers that describes a labeled tree. The code is constructed by following a process that removes $n - 2$ leaves from the tree. At each step, the leaf with the smallest label is removed, and the label of its only neighbor is added to the code.

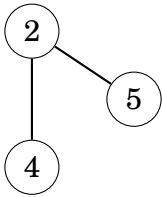
For example, let us calculate the Prüfer code of the following graph:



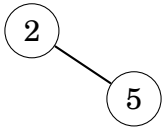
First we remove node 1 and add node 4 to the code:



Then we remove node 3 and add node 4 to the code:



Finally we remove node 4 and add node 2 to the code:



Thus, the Prüfer code of the graph is [4,4,2].

We can construct a Prüfer code for any tree, and more importantly, the original tree can be reconstructed from a Prüfer code. Hence, the number of labeled trees of n nodes equals n^{n-2} , the number of Prüfer codes of size n .

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。例えば、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。

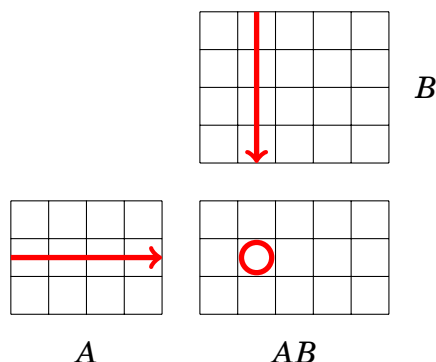
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

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。

Matrix multiplication is associative, so $A(BC) = (AB)C$ holds, but it is not commutative, so $AB = BA$ does not usually hold.

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。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

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。しかし、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

$$\det(A) = \sum_{j=1}^n A[1, j] C[1, j],$$

^{*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。

$C[i, j]$ は A における $[i, j]$ の余因子で次の様に示されます。

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

$M[i, j]$ 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、

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

The determinant of A tells us whether there is an **逆行列 - inverse matrix** A^{-1} such that $A \cdot A^{-1} = I$, where I is an identity matrix. It turns out that A^{-1} exists exactly when $\det(A) \neq 0$, and it can be calculated using the formula 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、次式で計算できます。

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

$$\underbrace{\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{pmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_I.$$

23.2 線型回帰 - Linear recurrences

線形漸化式 - 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

線形漸化式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 $O(\log n)$ 時間で計算できます。

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。

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

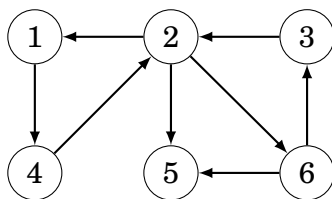
最初の $k-1$ 行は、1つの要素が1であることを除いて、各要素が0である。これらの行は、 $f(i)$ を $f(i+1)$ 、 $f(i+1)$ を $f(i+2)$ 、と置き換えていきます。最後の行には、新しい値 $f(i+k)$ を計算するための漸化式の係数が含まれます。さて、 $f(n)$ は次の式を用いて $O(k^3 \log n)$ 時間で計算できます。

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

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。

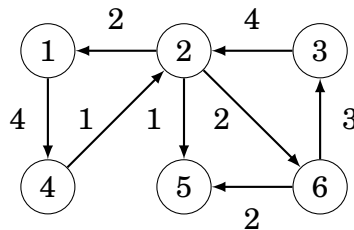
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

contains the numbers of paths of 4 edges between the nodes. 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て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。

$$AB[i,j] = \min_{k=1}^n A[i,k] + B[k,j]$$

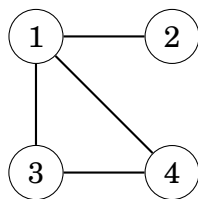
の代わりに最小値を、積の代わりに要素の和を計算します。これだけで、行列の累乗はグラフの最短経路に対応します。

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

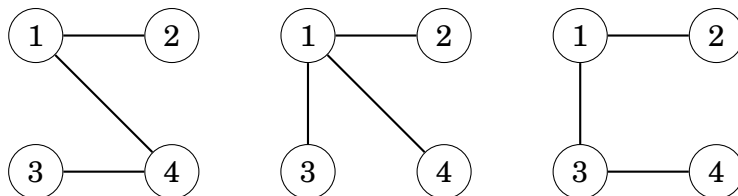
ノード 2 からノード 5 までの 4 辺のパスの最小長は 8 です。このようなパスは、 $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ 。

キルヒホッフの定理 - Kirchhoff's theorem

キルヒホッフの定理 - Kirchhoff's theorem 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、 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。

$$\det\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} = 3.$$

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。

$$\det\begin{bmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{bmatrix} = n^{n-2}.$$

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りう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、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。例えば、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 つのスートのうちから 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。このため、この確率は以下のようになります。

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

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も良いです。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。同様に、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ようになります。

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

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。例えば、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。そして、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。

$$P(A) = \sum_{x \in A} p(x).$$

例えば、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。
- **The 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。

$$P(\bar{A}) = 1 - P(A).$$

補集合を使うと、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。たとえば、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。このため、 $(5/6)^{10}$ は 10 投のうちただの一回も 1 投も 6 でない確率である。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、

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

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 の確率である。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 $1\dots3$ 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。

$$P(A|B) = P(A) \quad \text{and} \quad P(B|A) = P(B),$$

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、共通部分の確率は

$$P(A \cap B) = P(A)P(B).$$

例えば、山札からトランプを引く場合、

$$A = \text{"the suit is clubs"}$$

$$B = \text{"the value is four"}$$

は独立です。このため、

$$A \cap B = \text{"クラブの4である"}$$

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、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、 $P(X = 10) = 3/36$ 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、期待値は以下の通りです。

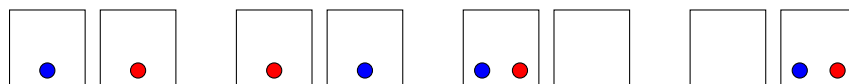
$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

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。

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

ここで、 n 個のボールが n 個の箱にランダムに入れられ、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

$$P(X=x) = (1-p)^{x-1}p,$$

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。

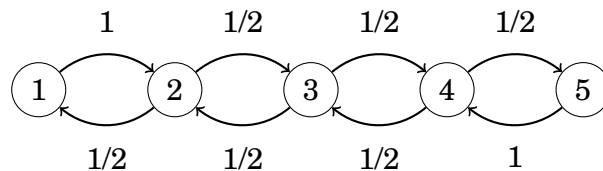
$$E[X] = \frac{1}{p}.$$

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、遷移をエッジとするグラフで表現することができます。

例として、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階下を歩きます。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 階にいる確率は何

この問題はマルコフ連鎖の1つで、 $n=5$ の場合、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。次

の様にします。

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

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 $[0, 1, 0, 0, 0]$ へは、次の様になります。

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

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。

A 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のことです。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。

次に、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？ 実は、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。

^{*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.

このアルゴリズムは、配列のランダムな要素 x を選んで、 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 個のとすると、 $O(n)$ 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。さて、 $a = k$ 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。

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、この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。しかし、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。しかし、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 の要素の X はランダムなベクトルを選びます。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、以下の様にします。 $A(BX)$ 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。

このアルゴリズムの欠点は、わずかな確率で間違いを犯す可能性があります。

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

ですが、

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

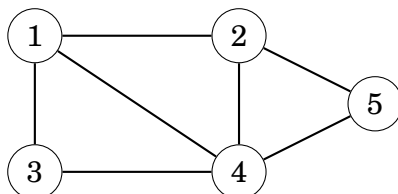
しかし、実際にはアルゴリズムが誤りを犯す確率は小さいので、 $AB = C$ と報告す

^{*2} R. M. Freivalds published this algorithm in 1977 [26], and it is sometimes called **Freivalds' algorithm**.

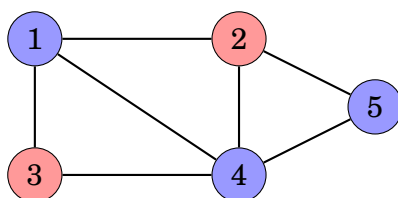
る前に、さらにいくつかのランダムベクトル X を用いて結果を検証することで確率を下げるすることができます。

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。



一例は



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。

一つの辺の端点が異なる色になる確率は $1/2$ なのですが、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。そして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ることがわかり、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。

For example, if $n = 10$, the game may proceed as follows:

- 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 $0, 1, 2, \dots, n$ の状態からなり、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である。これをを利用して、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)。

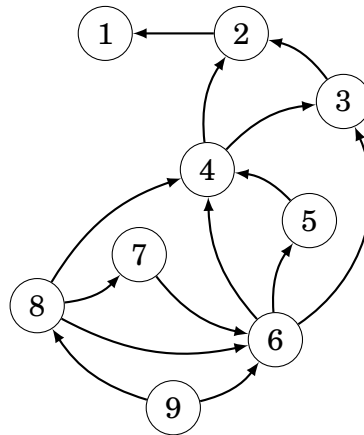
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| L | W | W | W | L | W | W | W | L | W | W | W | L | W | W | W |

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

ここで、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るが、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、まだ棒が入っている山を選び、そこから 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。例えば、 $[10, 12, 5]$ 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。

*¹ The optimal strategy for nim was published in 1901 by C. L. Bouton [10].

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。なぜなら、一つの値 x_k が変化すると、**nim sum** も変化するので、どんな作業の後も **nim sum** は 0 と異なるからです。

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き、 $x_k \oplus s$ 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 は s の左端の 1 ビットの 1 ビットを持った数です。

$[10,12,5]$ 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。

| | |
|----|------|
| 10 | 1010 |
| 12 | 1100 |
| 5 | 0101 |
| 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 |
| 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 |
| 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

帰るのは、次の手の後に各山が最大でも 1 本の棒を含むような状況で導入されることになる。標準的なゲームでは、1 本の棒を持つヒープが偶数個になるような手を選ぶべきである。

The idea is to first play the *misère* game like the standard game, but change the strategy at the end of the game. The new strategy will be introduced in a situation where each heap would contain at most one stick after the next move.

In the standard game, we should choose a move after which there is an even number of heaps with one stick. However, in the *misère* game, we choose a move so that there is an odd number of heaps with one stick.

This strategy works because a state where the strategy changes always appears in the game, and this state is a winning state, because it contains exactly one heap that has more than one stick so the nim sum is not 0.

25.3 Sprague – Grundy theorem

The **Sprague – Grundy theorem**^{*2} generalizes the strategy used in nim to all games that fulfil the following requirements:

- There are two players who move alternately.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

The idea is to calculate for each game state a Grundy number that corresponds to the number of sticks in a nim heap. When we know the Grundy numbers of all states, we can play the game like the nim game.

Grundy numbers

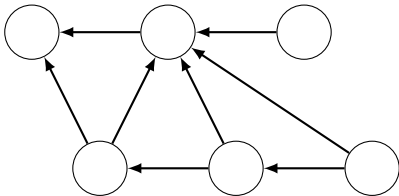
The **Grundy number** of a game state is

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

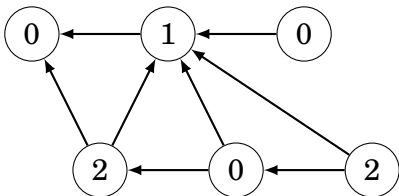
^{*2} The theorem was independently discovered by R. Sprague [61] and P. M. Grundy [31].

where g_1, g_2, \dots, g_n are the Grundy numbers of the states to which we can move, and the mex function gives the smallest nonnegative number that is not in the set. For example, $\text{mex}(\{0, 1, 3\}) = 2$. If there are no possible moves in a state, its Grundy number is 0, because $\text{mex}(\emptyset) = 0$.

For example, in the state graph



the Grundy numbers are as follows:

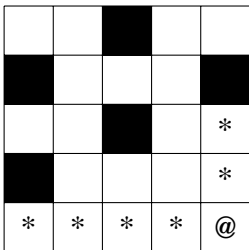


The Grundy number of a losing state is 0, and the Grundy number of a winning state is a positive number.

The Grundy number of a state corresponds to the number of sticks in a nim heap. If the Grundy number is 0, we can only move to states whose Grundy numbers are positive, and if the Grundy number is $x > 0$, we can move to states whose Grundy numbers include all numbers $0, 1, \dots, x - 1$.

As an example, consider a game where the players move a figure in a maze. Each square in the maze is either floor or wall. On each turn, the player has to move the figure some number of steps left or up. The winner of the game is the player who makes the last move.

The following picture shows a possible initial state of the game, where @ denotes the figure and * denotes a square where it can move.



The states of the game are all floor squares of the maze. In the above maze,

the Grundy numbers are as follows:

| | | | | |
|---|---|---|---|---|
| 0 | 1 | | 0 | 1 |
| | 0 | 1 | 2 | |
| 0 | 2 | | 1 | 0 |
| | 3 | 0 | 4 | 1 |
| 0 | 4 | 1 | 3 | 2 |

Thus, each state of the maze game corresponds to a heap in the nim game. For example, the Grundy number for the lower-right square is 2, so it is a winning state. We can reach a losing state and win the game by moving either four steps left or two steps up.

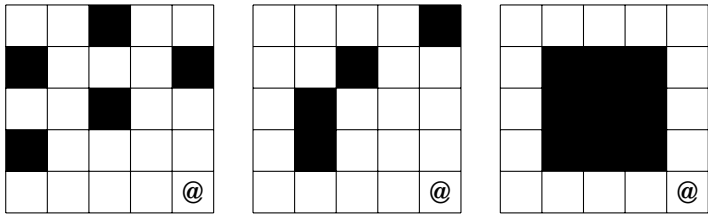
Note that unlike in the original nim game, it may be possible to move to a state whose Grundy number is larger than the Grundy number of the current state. However, the opponent can always choose a move that cancels such a move, so it is not possible to escape from a losing state.

Subgames

Next we will assume that our game consists of subgames, and on each turn, the player first chooses a subgame and then a move in the subgame. The game ends when it is not possible to make any move in any subgame.

In this case, the Grundy number of a game is the nim sum of the Grundy numbers of the subgames. The game can be played like a nim game by calculating all Grundy numbers for subgames and then their nim sum.

As an example, consider a game that consists of three mazes. In this game, on each turn, the player chooses one of the mazes and then moves the figure in the maze. Assume that the initial state of the game is as follows:



The Grundy numbers for the mazes are as follows:

| | | | | |
|---|---|---|---|---|
| 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 |

In the initial state, the nim sum of the Grundy numbers is $2 \oplus 3 \oplus 3 = 2$, so the first player can win the game. One optimal move is to move two steps up in the first maze, which produces the nim sum $0 \oplus 3 \oplus 3 = 0$.

Grundy’s game

Sometimes a move in a game divides the game into subgames that are independent of each other. In this case, the Grundy number of the game is

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

where n is the number of possible moves and

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

where move k generates subgames with Grundy numbers $a_{k,1}, a_{k,2}, \dots, a_{k,m}$.

An example of such a game is **Grundy’s game**. Initially, there is a single heap that contains n sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player who makes the last move wins the game.

Let $f(n)$ be the Grundy number of a heap that contains n sticks. The Grundy number can be calculated by going through all ways to divide the heap into two heaps. For example, when $n = 8$, the possibilities are $1 + 7$, $2 + 6$ and $3 + 5$, so

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

In this game, the value of $f(n)$ is based on the values of $f(1), \dots, f(n - 1)$. The base cases are $f(1) = f(2) = 0$, because it is not possible to divide the heaps of 1 and 2 sticks. The first Grundy numbers are:

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

The Grundy number for $n = 8$ is 2, so it is possible to win the game. The winning move is to create heaps $1 + 7$, because $f(1) \oplus f(7) = 0$.

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、 $O(n)$ または $O(n \log n)$ で動作するアルゴリズムが競技プログラミングでは求められることが多々あります。

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。例えば、ABC というパターンは、ABABCBABC という文字列の中に 2 回出現する。

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。ここは $O(n+m)$ 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。この時に、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.

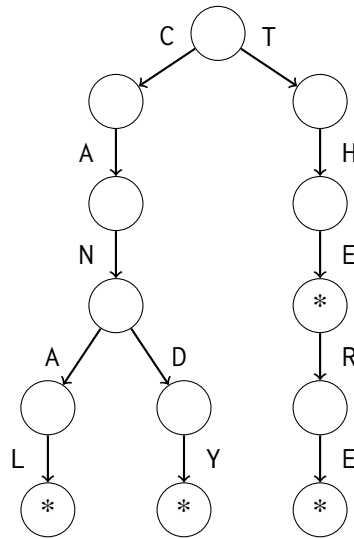
ピリオド - period とは文字列の接頭辞であって、そのピリオドを繰り返すことで文字列を構成することができるものです。最後の繰り返しは部分的で、ピリオドの接頭語だけを含むこともある。ABCABCA の最も短いピリオドは ABC です。(訳註: 最後の A のみの文字列はピリオドの 1 文字目で構成されています)

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。厳密には文字列が $x < y$ であるとは、その 2 つは異なることを前提として ($x \neq y$)、 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、 $O(n)$ 時間で可能です。

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、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、 $\text{trie}[s][c]$ 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** とは、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、時間の前処理を行った後は $O(1)$ 時間で計算することができる。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 を構築していきます。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、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ケースが多いです。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。このようなアルゴリズムの時間計算量は $O(n^2)$ です。

ブルートフォースアルゴリズムは文字列の部分文字列を比較する計算のため、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り、とても良い時間計算量となります。

また、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。これは、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。

Collisions are always possible, because the number of different strings is larger

than the number of different hash values. However, the probability of a collision is small if the constants A and B are carefully chosen. A usual way is to choose random constants near 10^9 , for example as follows:

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

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 型が使える。しかし、ハッシュ値は 10^9 くらいあれば十分なのではないでしょうか？

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、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ため、良いように思えます。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して 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範囲 $[x, y]$ 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 値を計算するときに

^{*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].

この情報を利用できます。

各位置 k で、まず $z[k-x]$ 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 $y+1$ で行われるためです。

例えば、次のような Z-array 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、 $s[0\dots 4]$ と $s[6\dots 10]$ 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 | ? | ? | ? | ? | ? | ? | ? |



しかし、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

| 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

A 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相性です。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、ある範囲の要素の総和を計算するために、範囲を 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て、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。

尚 \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、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 $|x_1 - x_2| + |y_1 - y_2|$ で定義します。

| | | | |
|---|---|---|---|
| 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である。このために、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、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 $O(n\sqrt{n})$ ですむことがわかります。これをもう少し見ていきます。

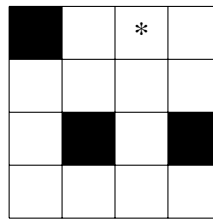
まず、文字 c に対してアルゴリズム 1 を使うとします。 \sqrt{n} 回以下出現する文字 c のセルと他のセルを比較します。この回数は最大で $O(n\sqrt{n})$ 回です。したがって、このようなすべてのセルの処理に使われる時間は $O(n\sqrt{n})$ です。

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。

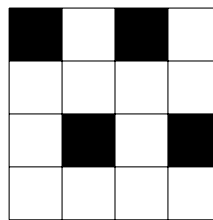
Batch processing

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するしたいと思います。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、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は、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 Algorithm 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 $O(\sqrt{n})$ 個の時間がかかる。

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。

という事実である。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く。今、 $1, 2, \dots, k$ という数を選ぶと、得られる総和は

$$\frac{k(k+1)}{2}.$$

になる。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ある。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、 $O(n\sqrt{n})$ 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る。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る。

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る。

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 を構築する。

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する。

文字列の長さは最大でも $O(\sqrt{m})$ 個なので、この結果、実行時間が $O(n\sqrt{m})$ のアルゴリズムとなる。

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範囲 (active range) を保持しておき、有効範囲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範囲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範囲の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

^{*1} According to [12]

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

配列の範囲に対応するクエリの集合が与えられて、範囲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 がアクティブ範囲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、アクティブな範囲は変更されます。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

Segment trees revisited

A segment tree is a versatile data structure that can be used to solve a large number of algorithm problems. However, there are many topics related to segment trees that we have not touched yet. Now is time to discuss some more advanced variants of segment trees.

So far, we have implemented the operations of a segment tree by walking *from bottom to top* in the tree. For example, we have calculated range sums as follows (Chapter 9.3):

```
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;
}
```

However, in more advanced segment trees, it is often necessary to implement the operations in another way, *from top to bottom*. Using this approach, the function becomes as follows:

```
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);
}
```

```
}

```

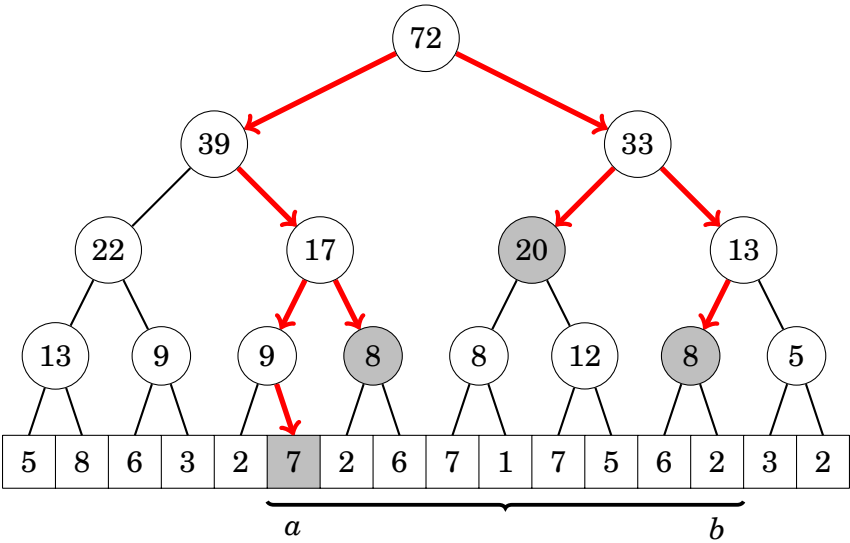
Now we can calculate any value of $\text{sum}_q(a, b)$ (the sum of array values in range $[a, b]$) as follows:

```
int s = sum(a, b, 1, 0, n-1);

```

The parameter k indicates the current position in tree. Initially k equals 1, because we begin at the root of the tree. The range $[x, y]$ corresponds to k and is initially $[0, n - 1]$. When calculating the sum, if $[x, y]$ is outside $[a, b]$, the sum is 0, and if $[x, y]$ is completely inside $[a, b]$, the sum can be found in tree. If $[x, y]$ is partially inside $[a, b]$, the search continues recursively to the left and right half of $[x, y]$. The left half is $[x, d]$ and the right half is $[d + 1, y]$ where $d = \lfloor \frac{x+y}{2} \rfloor$.

The following picture shows how the search proceeds when calculating the value of $\text{sum}_q(a, b)$. The gray nodes indicate nodes where the recursion stops and the sum can be found in tree.



Also in this implementation, operations take $O(\log n)$ time, because the total number of visited nodes is $O(\log n)$.

28.1 Lazy propagation

Using **lazy propagation**, we can build a segment tree that supports *both* range updates and range queries in $O(\log n)$ time. The idea is to perform updates and queries from top to bottom and perform updates *lazily* so that they are propagated down the tree only when it is necessary.

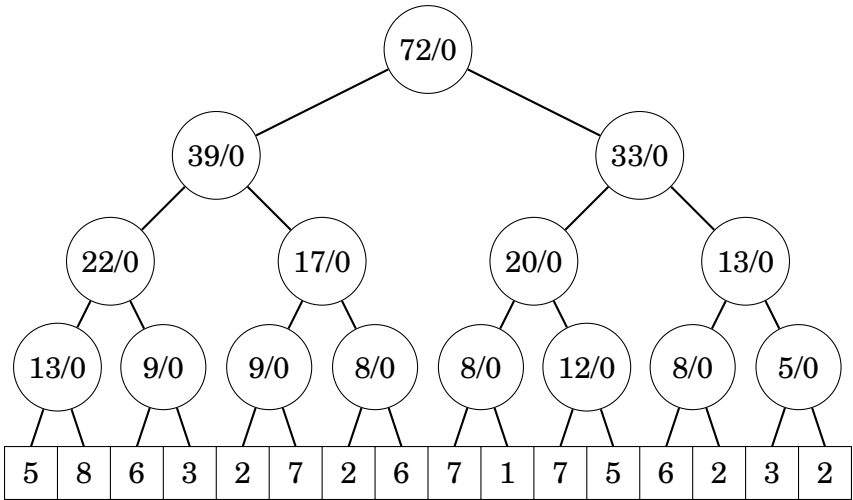
In a lazy segment tree, nodes contain two types of information. Like in an ordinary segment tree, each node contains the sum or some other value related to the corresponding subarray. In addition, the node may contain information related to lazy updates, which has not been propagated to its children.

There are two types of range updates: each array value in the range is either *increased* by some value or *assigned* some value. Both operations can be implemented using similar ideas, and it is even possible to construct a tree that supports both operations at the same time.

Lazy segment trees

Let us consider an example where our goal is to construct a segment tree that supports two operations: increasing each value in $[a, b]$ by a constant and calculating the sum of values in $[a, b]$.

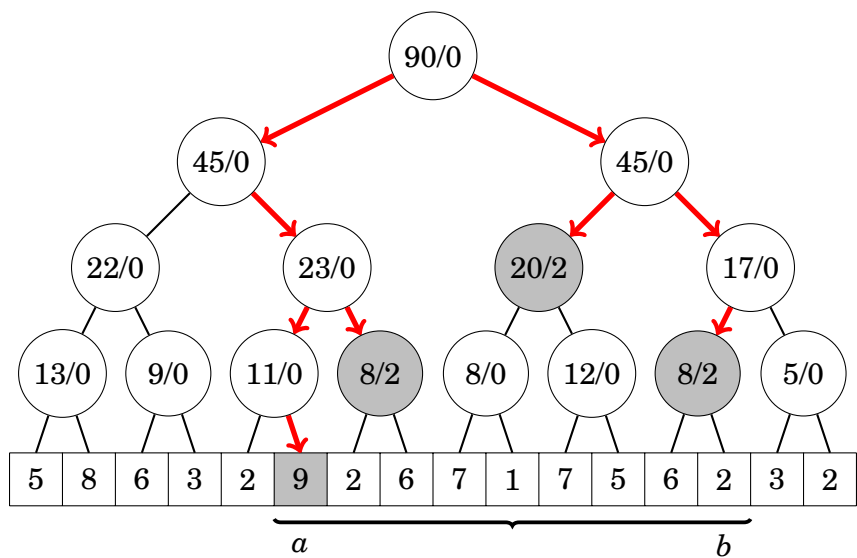
We will construct a tree where each node has two values s/z : s denotes the sum of values in the range, and z denotes the value of a lazy update, which means that all values in the range should be increased by z . In the following tree, $z = 0$ in all nodes, so there are no ongoing lazy updates.



When the elements in $[a, b]$ are increased by u , we walk from the root towards the leaves and modify the nodes of the tree as follows: If the range $[x, y]$ of a node is completely inside $[a, b]$, we increase the z value of the node by u and stop. If $[x, y]$ only partially belongs to $[a, b]$, we increase the s value of the node by hu , where h is the size of the intersection of $[a, b]$ and $[x, y]$, and continue our walk recursively in the tree.

For example, the following picture shows the tree after increasing the elements

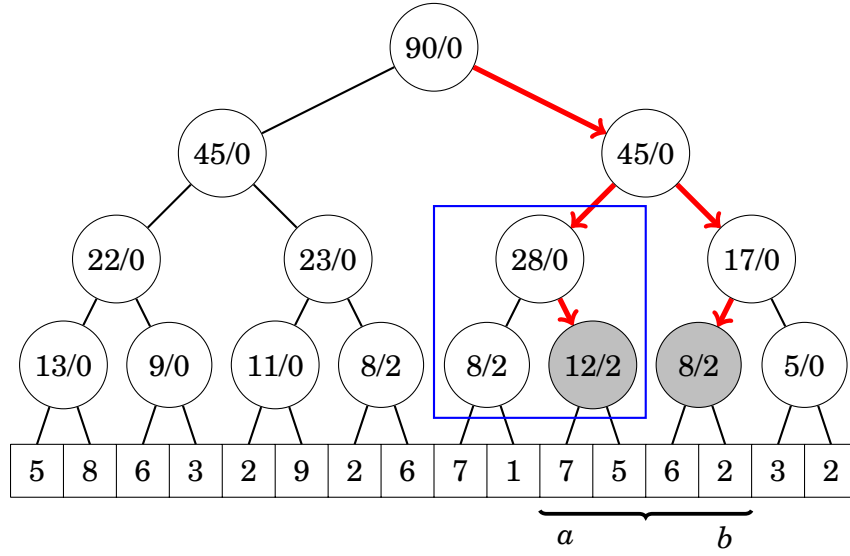
in $[a, b]$ by 2:



We also calculate the sum of elements in a range $[a, b]$ by walking in the tree from top to bottom. If the range $[x, y]$ of a node completely belongs to $[a, b]$, we add the s value of the node to the sum. Otherwise, we continue the search recursively downwards in the tree.

Both in updates and queries, the value of a lazy update is always propagated to the children of the node before processing the node. The idea is that updates will be propagated downwards only when it is necessary, which guarantees that the operations are always efficient.

The following picture shows how the tree changes when we calculate the value of $\text{sum}_a(a, b)$. The rectangle shows the nodes whose values change, because a lazy update is propagated downwards.



Note that sometimes it is needed to combine lazy updates. This happens when a node that already has a lazy update is assigned another lazy update. When calculating sums, it is easy to combine lazy updates, because the combination of updates z_1 and z_2 corresponds to an update $z_1 + z_2$.

Polynomial updates

Lazy updates can be generalized so that it is possible to update ranges using polynomials of the form

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0.$$

In this case, the update for a value at position i in $[a, b]$ is $p(i - a)$. For example, adding the polynomial $p(u) = u + 1$ to $[a, b]$ means that the value at position a increases by 1, the value at position $a + 1$ increases by 2, and so on.

To support polynomial updates, each node is assigned $k + 2$ values, where k equals the degree of the polynomial. The value s is the sum of the elements in the range, and the values z_0, z_1, \dots, z_k are the coefficients of a polynomial that corresponds to a lazy update.

Now, the sum of values in a range $[x, y]$ equals

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0.$$

The value of such a sum can be efficiently calculated using sum formulas. For example, the term z_0 corresponds to the sum $(y - x + 1)z_0$, and the term $z_1 u$

corresponds to the sum

$$z_1(0 + 1 + \dots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}.$$

When propagating an update in the tree, the indices of $p(u)$ change, because in each range $[x, y]$, the values are calculated for $u = 0, 1, \dots, y - x$. However, this is not a problem, because $p'(u) = p(u + h)$ is a polynomial of equal degree as $p(u)$. For example, if $p(u) = t_2u^2 + t_1u - t_0$, then

$$p'(u) = t_2(u + h)^2 + t_1(u + h) - t_0 = t_2u^2 + (2ht_2 + t_1)u + t_2h^2 + t_1h - t_0.$$

28.2 Dynamic trees

An ordinary segment tree is static, which means that each node has a fixed position in the array and the tree requires a fixed amount of memory. In a **dynamic segment tree**, memory is allocated only for nodes that are actually accessed during the algorithm, which can save a large amount of memory.

The nodes of a dynamic tree can be represented as structs:

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

Here `value` is the value of the node, `[x,y]` is the corresponding range, and `left` and `right` point to the left and right subtree.

After this, nodes can be created as follows:

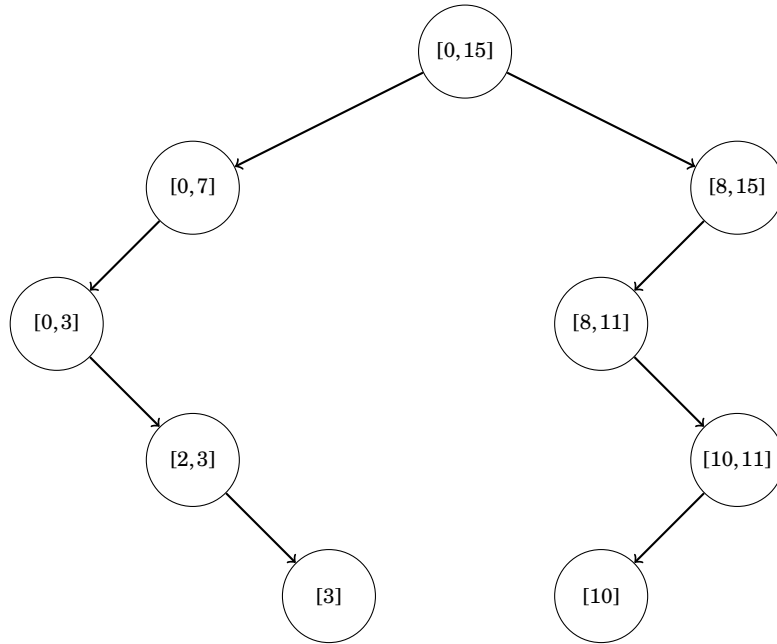
```
// create new node
node *x = new node(0, 0, 15);
// change value
x->value = 5;
```

Sparse segment trees

A dynamic segment tree is useful when the underlying array is *sparse*, i.e., the range $[0, n - 1]$ of allowed indices is large, but most array values are zeros. While an ordinary segment tree uses $O(n)$ memory, a dynamic segment tree only uses

$O(k \log n)$ memory, where k is the number of operations performed.

A **sparse segment tree** initially has only one node $[0, n - 1]$ whose value is zero, which means that every array value is zero. After updates, new nodes are dynamically added to the tree. For example, if $n = 16$ and the elements in positions 3 and 10 have been modified, the tree contains the following nodes:



Any path from the root node to a leaf contains $O(\log n)$ nodes, so each operation adds at most $O(\log n)$ new nodes to the tree. Thus, after k operations, the tree contains at most $O(k \log n)$ nodes.

Note that if we know all elements to be updated at the beginning of the algorithm, a dynamic segment tree is not necessary, because we can use an ordinary segment tree with index compression (Chapter 9.4). However, this is not possible when the indices are generated during the algorithm.

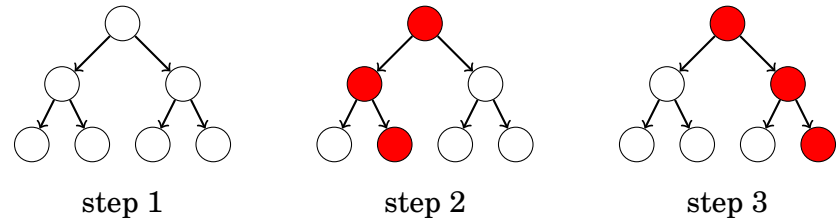
Persistent segment trees

Using a dynamic implementation, it is also possible to create a **persistent segment tree** that stores the *modification history* of the tree. In such an implementation, we can efficiently access all versions of the tree that have existed during the algorithm.

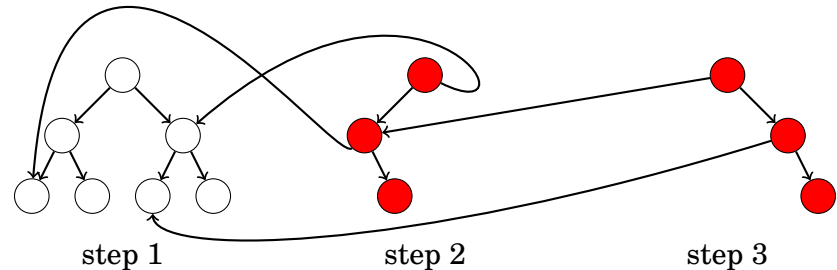
When the modification history is available, we can perform queries in any previous tree like in an ordinary segment tree, because the full structure of each tree is stored. We can also create new trees based on previous trees and modify

them independently.

Consider the following sequence of updates, where red nodes change and other nodes remain the same:



After each update, most nodes of the tree remain the same, so an efficient way to store the modification history is to represent each tree in the history as a combination of new nodes and subtrees of previous trees. In this example, the modification history can be stored as follows:



The structure of each previous tree can be reconstructed by following the pointers starting at the corresponding root node. Since each operation adds only $O(\log n)$ new nodes to the tree, it is possible to store the full modification history of the tree.

28.3 Data structures

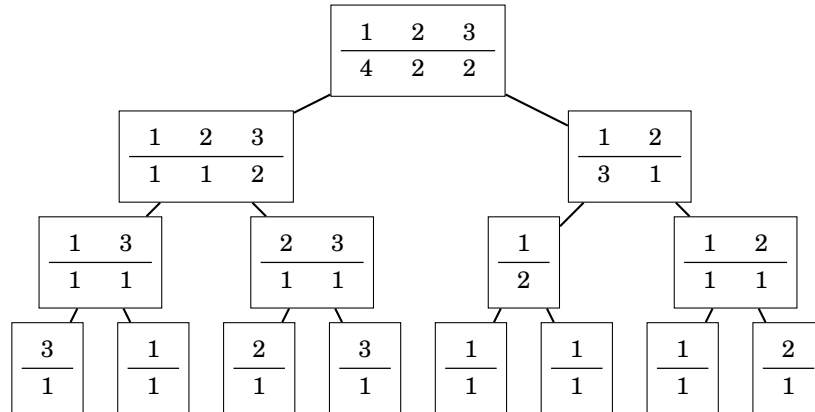
Instead of single values, nodes in a segment tree can also contain *data structures* that maintain information about the corresponding ranges. In such a tree, the operations take $O(f(n)\log n)$ time, where $f(n)$ is the time needed for processing a single node during an operation.

As an example, consider a segment tree that supports queries of the form "how many times does an element x appear in the range $[a, b]$?" For example, the element 1 appears three times in the following range:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 1 | 2 | 3 | 1 | 1 | 1 | 2 |
|---|---|---|---|---|---|---|---|

To support such queries, we build a segment tree where each node is assigned a data structure that can be asked how many times any element x appears in the corresponding range. Using this tree, the answer to a query can be calculated by combining the results from the nodes that belong to the range.

For example, the following segment tree corresponds to the above array:



We can build the tree so that each node contains a map structure. In this case, the time needed for processing each node is $O(\log n)$, so the total time complexity of a query is $O(\log^2 n)$. The tree uses $O(n \log n)$ memory, because there are $O(\log n)$ levels and each level contains $O(n)$ elements.

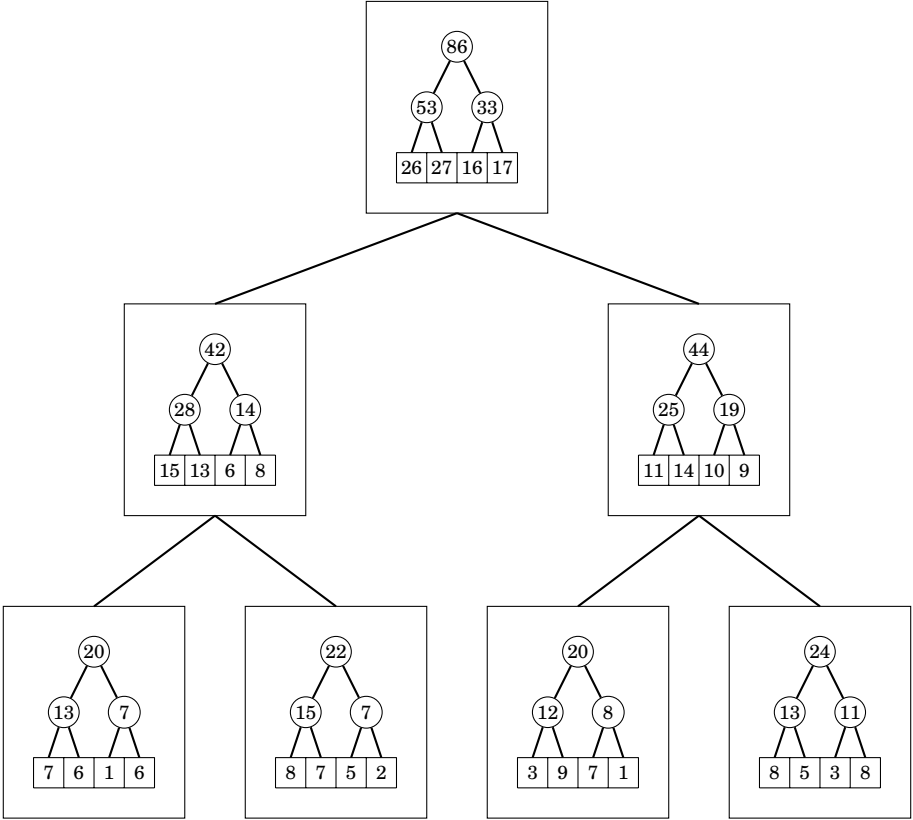
28.4 Two-dimensionality

A **two-dimensional segment tree** supports queries related to rectangular subarrays of a two-dimensional array. Such a tree can be implemented as nested segment trees: a big tree corresponds to the rows of the array, and each node contains a small tree that corresponds to a column.

For example, in the array

| | | | |
|---|---|---|---|
| 7 | 6 | 1 | 6 |
| 8 | 7 | 5 | 2 |
| 3 | 9 | 7 | 1 |
| 8 | 5 | 3 | 8 |

the sum of any subarray can be calculated from the following segment tree:



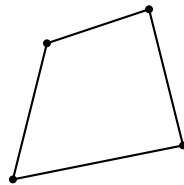
The operations of a two-dimensional segment tree take $O(\log^2 n)$ time, because the big tree and each small tree consist of $O(\log n)$ levels. The tree requires $O(n^2)$ memory, because each small tree contains $O(n)$ values.

第 29 章

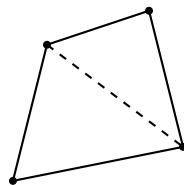
Geometry

In geometric problems, it is often challenging to find a way to approach the problem so that the solution to the problem can be conveniently implemented and the number of special cases is small.

As an example, consider a problem where we are given the vertices of a quadrilateral (a polygon that has four vertices), and our task is to calculate its area. For example, a possible input for the problem is as follows:



One way to approach the problem is to divide the quadrilateral into two triangles by a straight line between two opposite vertices:



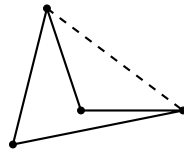
After this, it suffices to sum the areas of the triangles. The area of a triangle can be calculated, for example, using **Heron's formula**

$$\sqrt{s(s-a)(s-b)(s-c)},$$

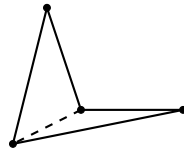
where a , b and c are the lengths of the triangle's sides and $s = (a + b + c)/2$.

This is a possible way to solve the problem, but there is one pitfall: how to divide the quadrilateral into triangles? It turns out that sometimes we cannot

just pick two arbitrary opposite vertices. For example, in the following situation, the division line is *outside* the quadrilateral:



However, another way to draw the line works:



It is clear for a human which of the lines is the correct choice, but the situation is difficult for a computer.

However, it turns out that we can solve the problem using another method that is more convenient to a programmer. Namely, there is a general formula

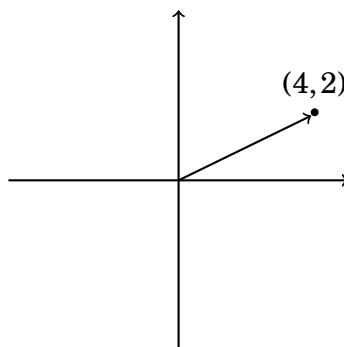
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

that calculates the area of a quadrilateral whose vertices are (x_1, y_1) , (x_2, y_2) , (x_3, y_3) and (x_4, y_4) . This formula is easy to implement, there are no special cases, and we can even generalize the formula to *all* polygons.

29.1 Complex numbers

A **complex number** is a number of the form $x + yi$, where $i = \sqrt{-1}$ is the **imaginary unit**. A geometric interpretation of a complex number is that it represents a two-dimensional point (x, y) or a vector from the origin to a point (x, y) .

For example, $4 + 2i$ corresponds to the following point and vector:



The C++ complex number class `complex` is useful when solving geometric problems. Using the class we can represent points and vectors as complex numbers, and the class contains tools that are useful in geometry.

In the following code, `C` is the type of a coordinate and `P` is the type of a point or a vector. In addition, the code defines macros `X` and `Y` that can be used to refer to `x` and `y` coordinates.

```
typedef long long C;
typedef complex<C> P;
#define X real()
#define Y imag()
```

For example, the following code defines a point $p = (4, 2)$ and prints its `x` and `y` coordinates:

```
P p = {4, 2};
cout << p.X << " " << p.Y << "\n"; // 4 2
```

The following code defines vectors $v = (3, 1)$ and $u = (2, 2)$, and after that calculates the sum $s = v + u$.

```
P v = {3, 1};
P u = {2, 2};
P s = v + u;
cout << s.X << " " << s.Y << "\n"; // 5 3
```

In practice, an appropriate coordinate type is usually `long long` (integer) or `long double` (real number). It is a good idea to use integer whenever possible, because calculations with integers are exact. If real numbers are needed, precision errors should be taken into account when comparing numbers. A safe way to check if real numbers a and b are equal is to compare them using $|a - b| < \epsilon$, where ϵ is a small number (for example, $\epsilon = 10^{-9}$).

Functions

In the following examples, the coordinate type is `long double`.

The function `abs(v)` calculates the length $|v|$ of a vector $v = (x, y)$ using the formula $\sqrt{x^2 + y^2}$. The function can also be used for calculating the distance between points (x_1, y_1) and (x_2, y_2) , because that distance equals the length of the vector $(x_2 - x_1, y_2 - y_1)$.

The following code calculates the distance between points (4,2) and (3,-1):

```
P a = {4,2};
P b = {3,-1};
cout << abs(b-a) << "\n"; // 3.16228
```

The function $\arg(v)$ calculates the angle of a vector $v = (x, y)$ with respect to the x axis. The function gives the angle in radians, where r radians equals $180r/\pi$ degrees. The angle of a vector that points to the right is 0, and angles decrease clockwise and increase counterclockwise.

The function $\text{polar}(s, a)$ constructs a vector whose length is s and that points to an angle a . A vector can be rotated by an angle a by multiplying it by a vector with length 1 and angle a .

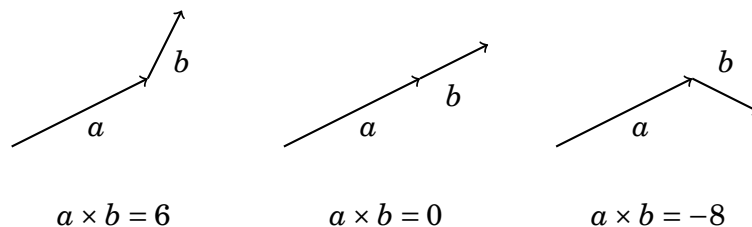
The following code calculates the angle of the vector (4,2), rotates it $1/2$ radians counterclockwise, and then calculates the angle again:

```
P v = {4,2};
cout << arg(v) << "\n"; // 0.463648
v *= polar(1.0, 0.5);
cout << arg(v) << "\n"; // 0.963648
```

29.2 Points and lines

The **cross product** $a \times b$ of vectors $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is calculated using the formula $x_1y_2 - x_2y_1$. The cross product tells us whether b turns left (positive value), does not turn (zero) or turns right (negative value) when it is placed directly after a .

The following picture illustrates the above cases:



For example, in the first case $a = (4, 2)$ and $b = (1, 2)$. The following code calculates the cross product using the class `complex`:

```
P a = {4,2};
```

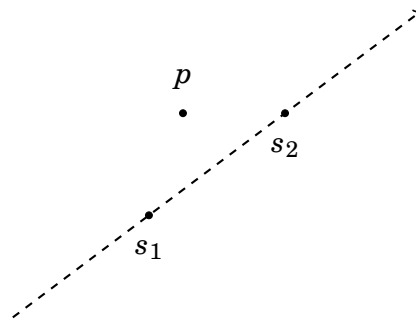
```
P b = {1,2};
C p = (conj(a)*b).Y; // 6
```

The above code works, because the function `conj` negates the y coordinate of a vector, and when the vectors $(x_1, -y_1)$ and (x_2, y_2) are multiplied together, the y coordinate of the result is $x_1y_2 - x_2y_1$.

Point location

Cross products can be used to test whether a point is located on the left or right side of a line. Assume that the line goes through points s_1 and s_2 , we are looking from s_1 to s_2 and the point is p .

For example, in the following picture, p is on the left side of the line:

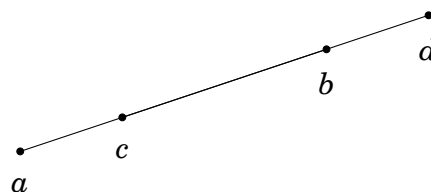


The cross product $(p - s_1) \times (p - s_2)$ tells us the location of the point p . If the cross product is positive, p is located on the left side, and if the cross product is negative, p is located on the right side. Finally, if the cross product is zero, points s_1 , s_2 and p are on the same line.

Line segment intersection

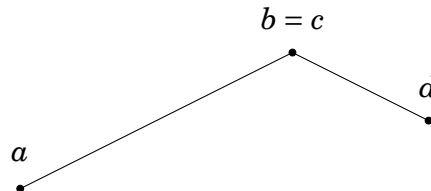
Next we consider the problem of testing whether two line segments ab and cd intersect. The possible cases are:

Case 1: The line segments are on the same line and they overlap each other. In this case, there is an infinite number of intersection points. For example, in the following picture, all points between c and b are intersection points:



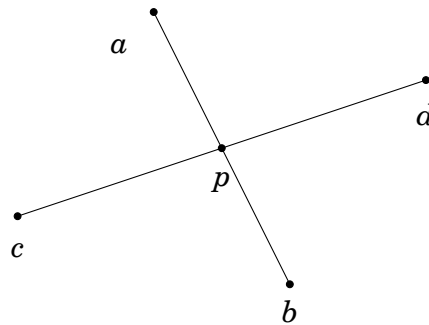
In this case, we can use cross products to check if all points are on the same line. After this, we can sort the points and check whether the line segments overlap each other.

Case 2: The line segments have a common vertex that is the only intersection point. For example, in the following picture the intersection point is $b = c$:



This case is easy to check, because there are only four possibilities for the intersection point: $a = c$, $a = d$, $b = c$ and $b = d$.

Case 3: There is exactly one intersection point that is not a vertex of any line segment. In the following picture, the point p is the intersection point:



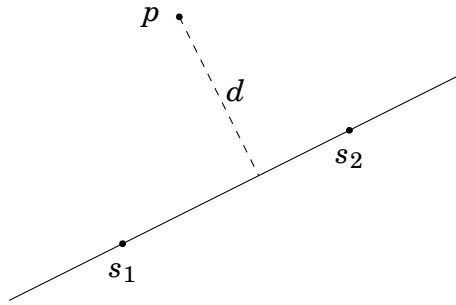
In this case, the line segments intersect exactly when both points c and d are on different sides of a line through a and b , and points a and b are on different sides of a line through c and d . We can use cross products to check this.

Point distance from a line

Another feature of cross products is that the area of a triangle can be calculated using the formula

$$\frac{|(a - c) \times (b - c)|}{2},$$

where a , b and c are the vertices of the triangle. Using this fact, we can derive a formula for calculating the shortest distance between a point and a line. For example, in the following picture d is the shortest distance between the point p and the line that is defined by the points s_1 and s_2 :

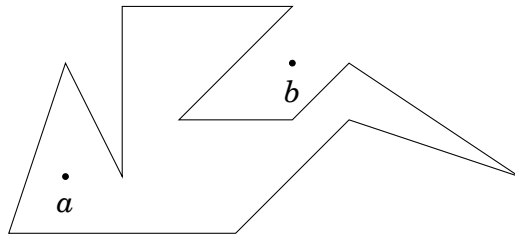


The area of the triangle whose vertices are s_1 , s_2 and p can be calculated in two ways: it is both $\frac{1}{2}|s_2 - s_1|d$ and $\frac{1}{2}((s_1 - p) \times (s_2 - p))$. Thus, the shortest distance is

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

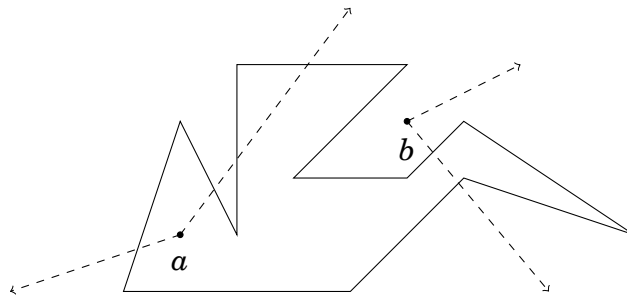
Point inside a polygon

Let us now consider the problem of testing whether a point is located inside or outside a polygon. For example, in the following picture point a is inside the polygon and point b is outside the polygon.



A convenient way to solve the problem is to send a *ray* from the point to an arbitrary direction and calculate the number of times it touches the boundary of the polygon. If the number is odd, the point is inside the polygon, and if the number is even, the point is outside the polygon.

For example, we could send the following rays:



The rays from a touch 1 and 3 times the boundary of the polygon, so a is inside the polygon. Correspondingly, the rays from b touch 0 and 2 times the boundary

of the polygon, so b is outside the polygon.

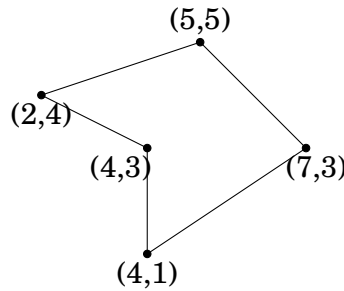
29.3 Polygon area

A general formula for calculating the area of a polygon, sometimes called the **shoelace formula**, is as follows:

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

Here the vertices are $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ in such an order that p_i and p_{i+1} are adjacent vertices on the boundary of the polygon, and the first and last vertex is the same, i.e., $p_1 = p_n$.

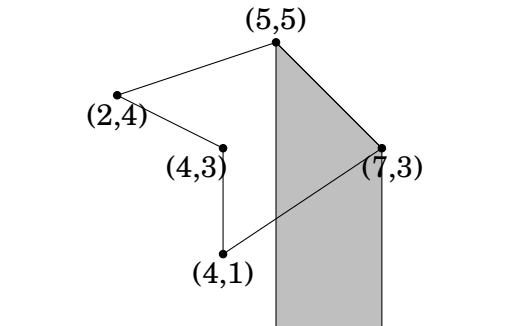
For example, the area of the polygon



is

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

The idea of the formula is to go through trapezoids whose one side is a side of the polygon, and another side lies on the horizontal line $y = 0$. For example:



The area of such a trapezoid is

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

where the vertices of the polygon are p_i and p_{i+1} . If $x_{i+1} > x_i$, the area is positive, and if $x_{i+1} < x_i$, the area is negative.

The area of the polygon is the sum of areas of all such trapezoids, which yields the formula

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

Note that the absolute value of the sum is taken, because the value of the sum may be positive or negative, depending on whether we walk clockwise or counterclockwise along the boundary of the polygon.

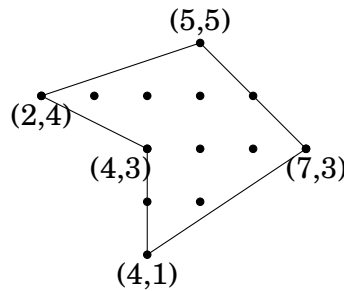
Pick's theorem

Pick's theorem provides another way to calculate the area of a polygon provided that all vertices of the polygon have integer coordinates. According to Pick's theorem, the area of the polygon is

$$a + b/2 - 1,$$

where a is the number of integer points inside the polygon and b is the number of integer points on the boundary of the polygon.

For example, the area of the polygon



is $6 + 7/2 - 1 = 17/2$.

29.4 Distance functions

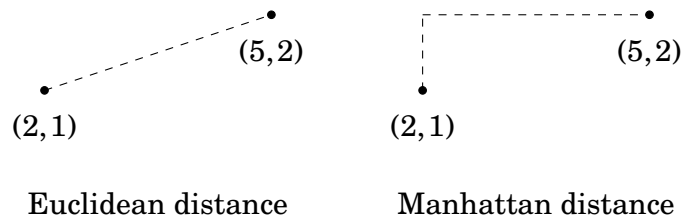
A **distance function** defines the distance between two points. The usual distance function is the **Euclidean distance** where the distance between points (x_1, y_1) and (x_2, y_2) is

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

An alternative distance function is the **Manhattan distance** where the distance between points (x_1, y_1) and (x_2, y_2) is

$$|x_1 - x_2| + |y_1 - y_2|.$$

For example, consider the following picture:



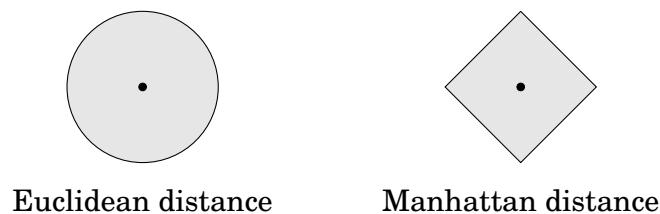
The Euclidean distance between the points is

$$\sqrt{(5-2)^2 + (2-1)^2} = \sqrt{10}$$

and the Manhattan distance is

$$|5-2| + |2-1| = 4.$$

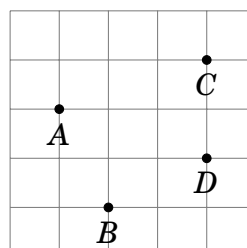
The following picture shows regions that are within a distance of 1 from the center point, using the Euclidean and Manhattan distances:



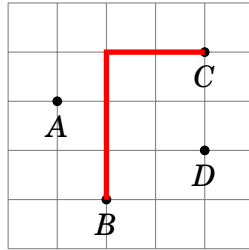
Rotating coordinates

Some problems are easier to solve if Manhattan distances are used instead of Euclidean distances. As an example, consider a problem where we are given n points in the two-dimensional plane and our task is to calculate the maximum Manhattan distance between any two points.

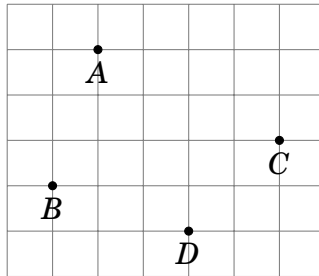
For example, consider the following set of points:



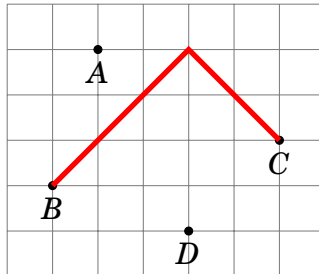
The maximum Manhattan distance is 5 between points B and C :



A useful technique related to Manhattan distances is to rotate all coordinates 45 degrees so that a point (x, y) becomes $(x + y, y - x)$. For example, after rotating the above points, the result is:



And the maximum distance is as follows:



Consider two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ whose rotated coordinates are $p'_1 = (x'_1, y'_1)$ and $p'_2 = (x'_2, y'_2)$. Now there are two ways to express the Manhattan distance between p_1 and p_2 :

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

For example, if $p_1 = (1, 0)$ and $p_2 = (3, 3)$, the rotated coordinates are $p'_1 = (1, -1)$ and $p'_2 = (6, 0)$ and the Manhattan distance is

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

The rotated coordinates provide a simple way to operate with Manhattan distances, because we can consider x and y coordinates separately. To maximize

the Manhattan distance between two points, we should find two points whose rotated coordinates maximize the value of

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|).$$

This is easy, because either the horizontal or vertical difference of the rotated coordinates has to be maximum.

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。

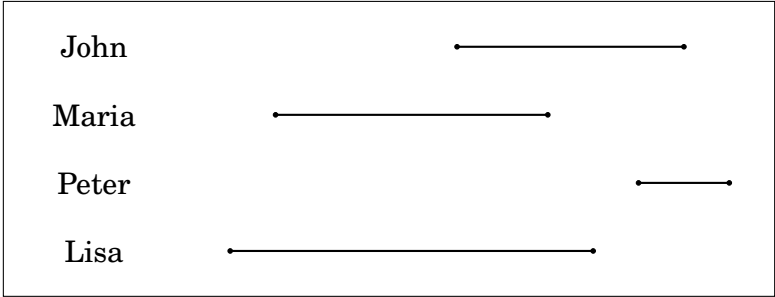
例えば、従業員数 n 人の会社があり、各従業員のある日の出社・退社時刻がわかっているとします。我々の仕事は、同時にオフィスにいた従業員の最大人数を計算することです。

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。

例を示します。

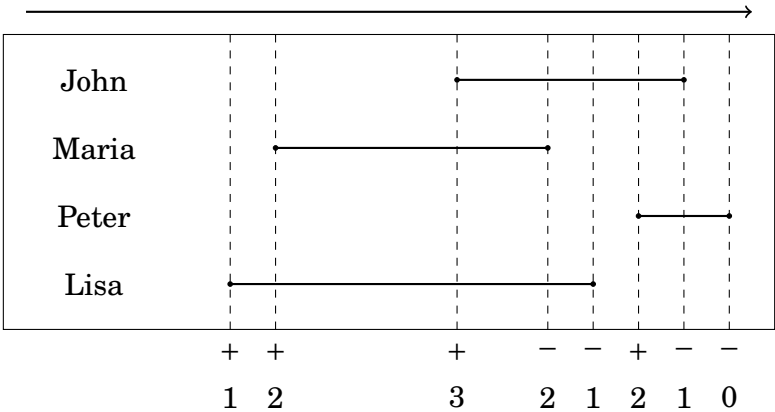
| 名前 | 出社 | 退社 |
|-------|----|----|
| John | 10 | 15 |
| Maria | 6 | 12 |
| Peter | 14 | 16 |
| Lisa | 5 | 13 |

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辿り、カウンタを管理します。出社イベントの時、カウンタをインクリメントします。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。

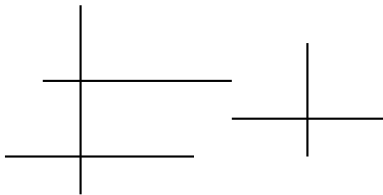


各記号はカウンターの値が増加するか減少するかを示し、カウンターの値は記号の下に示すようになります。カウンターの最大値は、ジョンが到着してからマリアが帰るまでの間の3です。

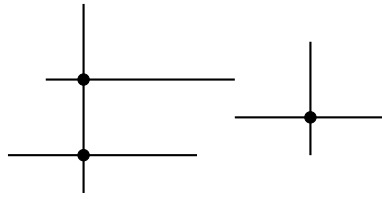
イベントのソートに $O(n \log n)$ の時間がかかり、残りのアルゴリズムに $O(n)$ の時間がかかるため、このアルゴリズムの実行時間は $O(n \log n)$ である。

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。



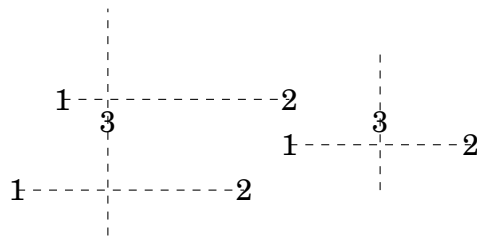
この場合の交差点は3つです。



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、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。



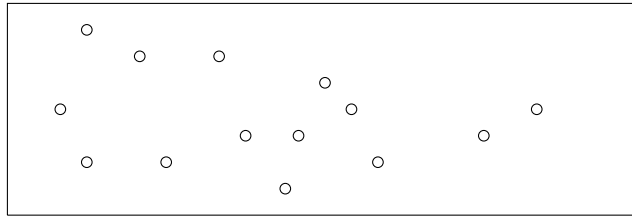
イベントを左から右へみていき、アクティブな水平線が存在する y 座標のセットを保持するデータ構造を持ちます。イベント 1 では、セグメントの y 座標をセットに追加し、イベント 2 では、その y 座標をセットから削除します。

イベント 3 を処理する時に交点を計算します。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、この数を交点の総数に加えればよいです。

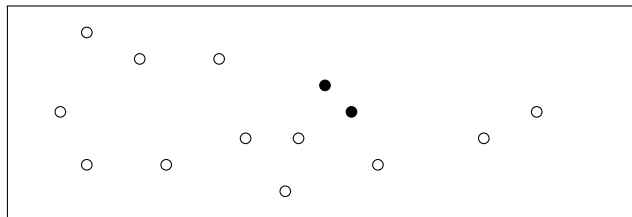
水平線の y 座標を格納するために、BIT または セグメントツリーを使用し、場合によっては座標圧縮を行います。このような構造を使用する場合、各イベントの処理に $O(\log n)$ の時間がかかるため、アルゴリズムの総実行時間は $O(n \log n)$ となります。

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



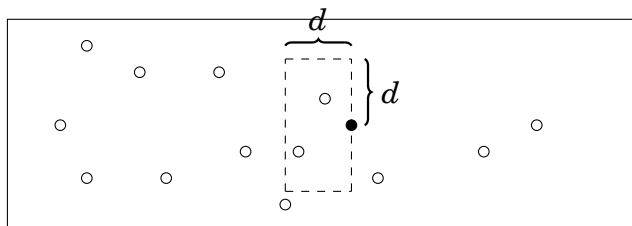
のとき、以下の2点を見つけます。



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。左から右へと点を巡り、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。そのため、これらの範囲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 以内の距離に存在しう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。

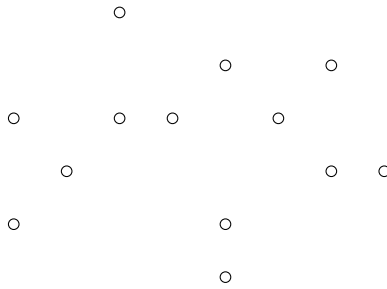
このアルゴリズムの時間計算量は $O(n \log n)$ です。 n の各点について左に最も近い点を $O(\log n)$ 時間で求められます。

*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.

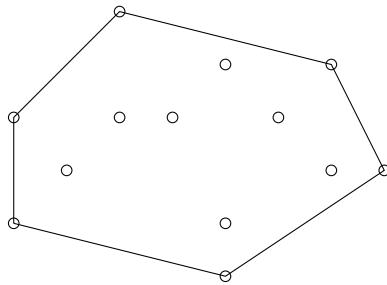
30.3 凸包問題 - Convex hull problem

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。

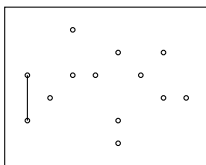


この時の凸包は

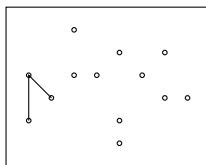


アンドリューのアルゴリズム - Andrew's algorithm [3] は、点群に対する凸包を $O(n \log n)$ 時間構成するアルゴリズムです。このアルゴリズムでは、まず左端と右端の点を求め、凸包を上包と下包の 2 つに分けます。このとき、まず上部の構築に集中します。

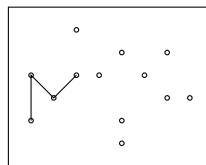
まず、点を主に x 座標で、次に y 座標の順で並べ替えます。その後、各ポイントを凸包に追加していきます。点を追加した後は、常に、凸包の最後の線分が左に曲がらないようにします。左に曲がっているならば、最後から 2 つ目の点を凸包から取り除いていきます。以下の図は、**Andrew** のアルゴリズムがどのように動作するかを示しています。



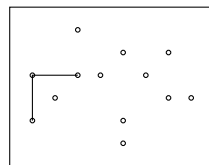
1



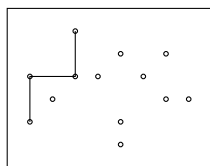
2



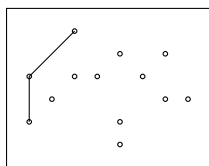
3



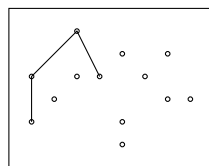
4



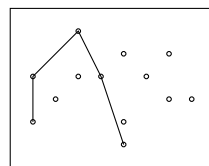
5



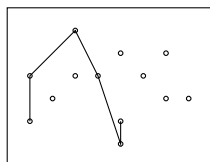
6



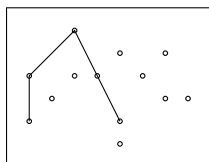
7



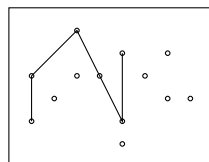
8



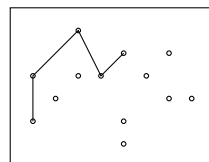
9



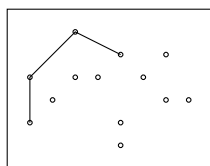
10



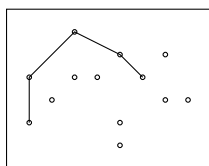
11



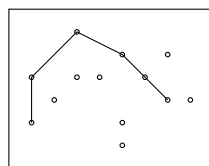
12



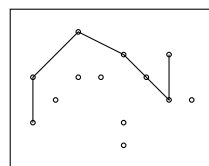
13



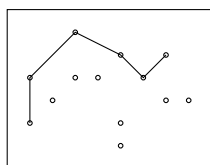
14



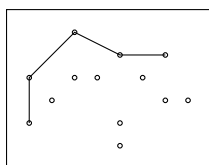
15



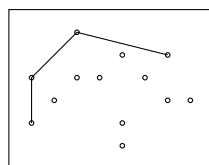
16



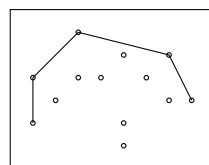
17



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.

