好きなアルゴリズムとデータ構造について

InTheBloom [著]

好きなアルゴリズムと
データ構造について

［著］InTheBloom

UEComic! 9

2025 年 11 月 13 日　発行

まえがき

本書を手に取って頂きありがとうございます。本書は私が競技プログラミングをやっている中で、特に面白いなと思ったアルゴリズムとデータ構造を紹介する本です。

基本的に自己満足で書いたため、議論の厳密さやわかりやすさなどは「私が満足できるもの」という基準に従っています。掲載したプログラム例は私が普段使っている D 言語[*1]を採用しましたが、D 特有の機能の使用は必要がなければ避けています。

本書に掲載されたプログラムはすべて https://github.com/InTheBloom/UEComic9 で公開しています。誤りなどがありましたら私の X アカウント（https://x.com/uu9782wsedandhp）か gmail（nato.rider.smm2@gmail.com）まで連絡いただけると幸いです。

それではアルゴリズムとデータ構造の世界をお楽しみください。

[*1] https://dlang.org/

目次

第 1 章

二分探索

本章では、二分探索というアルゴリズムを具体例から一般化しながら見ていきます。二分探索は簡単なようで実装を誤りやすく、添え字のずれによるバグ（off-by-one）が起こりやすいアルゴリズムでもあります。そうした誤りを避けるための考え方を段階的に整理していきます。章の最後には、使いやすく添え字ミスしにくい一般化二分探索を紹介します。

1.1 レベル 1: ソート済みの配列に対する検索

本節では、ソート済みの配列から特定の値を検索するためのアルゴリズムとして二分探索を導入し、その動作を説明します。

1.1.1 問題と解法

次の問題を考えます。

.....

長さ N の整数配列 A が与えられる。A は昇順ソート済みである。また、値 x が与えられる。ただし、A は x と等しい要素を少なくとも一つ含む。

A 中での x の場所、つまり $A[i] = x$ なる i を求めよ。

.....

この問題は、配列を先頭から順に見ていくことで $O(N)$ 時間で解くことができますが、配列がソート済みであることを利用するとより高速に解くことができます。

例えば配列 A の p 番目の要素が x 未満だったとしましょう。すると A は昇順ソートされていますから、目的の値 x は p 番目以降にあることがわかります。逆に、そうでない場合は x は p 番目より前にあることがわかります。

このように、たった一か所の値を確認するだけで、「配列全体から x を探す問題」を「p 番目以降から x を探す問題」か「p 番目より前から x を探す問題」のどちらかにすることができます。これを繰り返し適用することで非常に高速に問題を解くことができます。

p をどうとっても動作しますが、考えている配列長 N のちょうど中間である $\lfloor N/2 \rfloor$ などを取ると、x が p 番目より大きくても小さくても必ず問題サイズを約半分にできます。1 ステップごとに問題サイズを半分にしていくので「二分」探索と呼ばれています。

このアルゴリズムの時間計算量を評価しましょう。この手順 1 回につき問題サイズ N は必ず $1/2$ 倍以下になります。従って、問題を解くまでに必要な手順の回数 M は、$N/2^M \leq 1$ なる最小の M と同程度であると評価できます。以上より、時間計算量は $O(\log N)$ です。

ただし、あらかじめ A についての情報がわかっている場合はさらに最適な p の取り方がある場合があります。[*1]

1.1.2　プログラム例

以上のアイデアをもとに配列 A と整数 x を受け取り、x と等しい要素があるインデックスを返却する関数 `levelOne` を以下に示します。プログラム中の `l` と `r` は x を 1 つ以上含む範囲 $[l, r]$ を表しています。

```
int levelOne (int[] A, int x) {
    const int N = cast(int)(A.length);
    assert(0 < N);
    if (N == 1) {
        return 0;
    }

    int l = 0;
    int r = N - 1;

    while (1 < r - l) {
        int m = (l + r) / 2;
        if (A[m] < x) {
```

[*1] ほとんどの場合、このような工夫をしてもオーダーの改善にはなりません。

```
            l = m;
        }
        else {
            r = m;
        }
    }

    // A[l]かA[r]がxと等しい
    if (A[l] == x) {
        return l;
    }
    return r;
}
```

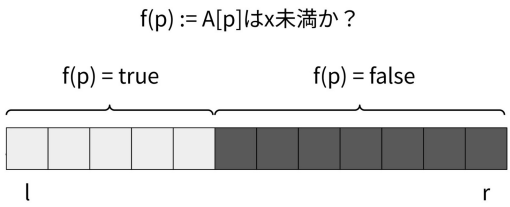
1.2　レベル 2: 単調な関数の境界探索

本節では、レベル 1 の問題に対する解法を一般化し、全く別の問題にも二分探索が適用できることを説明します。

1.2.1　一般化

前節で扱った検索問題は、次の 2 ステップで解いたと一般化できます。

1. $0 \leq p \leq N-1$ なる整数 p それぞれに、「$A[p]$ の値が x 未満かどうか」という真理値を対応させる写像 $f : \mathbb{N} \to \{\mathrm{true}, \mathrm{false}\}$ を用意する。
2. $0 \leq i \leq N-2$ かつ $f(i) \neq f(i+1)$ を満たす i を探す。



▲図 1.1: 写像 f の値の境界

つまり、二分探索とは終集合が二値である写像の境界地点を探すアルゴリズムであるといえます。図 1.1 は $f(p) := A[p]$ は x 未満か に対する境界を図示したものです。$A[i] = x$ なる i は（存在するなら）false 側の先頭の場所になるはずです。

配列の検索のように、f の値が切り替わる場所がちょうどひとつの場合は必ずその場所を見つけることができます。しかし、切り替わる場所が複数あったとしても、二分探索を適用する時点で $f(l) \neq f(r)$ でさえあれば切り替わる場所を少なくとも一つ必ず見つけることができます。図 1.2 の分布は切り替わる場所が複数ありますが、$f(l) \neq f(r)$ であるため問題なく境界を見つけることができます。[*2]

▲図 1.2: 検索可能な分布の例

このように一般化することで、適切な f さえ用意できれば同様のアルゴリズムを異なる問題へ適用することができます。例として、次の問題を考えてみましょう。

.....

$998244353 \leq x^2$ なる最小の整数 x を求めよ。

.....

一度 $998244353 \leq x^2$ となってしまえば、x 以上の数において不等号はずっとそのままです。従って、その切り替わる場所は一意であり、二分探索を適用することができます。

x は明らかに $1 \leq x \leq 998244353$ を満たすので、探索範囲は $l = 1, r = 998244353$ としておきます。$f(p) := p^2 < 998244353$ か？ とすると、true 側は二乗してもすべて 998244353 未満、false 側はすべて 998244353 超過となるため、この f を用いて探索し、false 側の先頭をみればよいです。

1.2.2　プログラム例

以上のアイデアをもとに整数 l、整数 r、関数 f を受け取り、$l \leq i \leq r - 1$ かつ $f(i) \neq f(i + 1)$ なる整数 i を返却する関数 `levelTwo` を以下に示します。ただし、

[*2] しかし、競技プログラミングにおいて切り替わりが複数ある問題の割合は多くありません。

$l = r$ である場合は l を返すものとし、そうでない場合は $f(l) \neq f(r)$ の事前条件を課すことにします。

```
int levelTwo (int l, int r, bool delegate(int) f) {
    assert(l <= r);
    if (l == r) {
        return l;
    }
    assert(f(l) != f(r));

    while (1 < r - l) {
        int m = (l + r) / 2;
        if (f(m)) {
            l = m;
        }
        else {
            r = m;
        }
    }

    if (f(l)) {
        return r;
    }
    return l;
}
```

1.3　レベル 3: 区間の分割としての二分探索

レベル 2 の一般化は非常に強力で、これだけでほとんどの二分探索を用いる問題に対応できます。ですがまだ改善の余地があります。

レベル 3 ではレベル 2 の二分探索で扱いきれない問題への対応と、使用者が off-by-one エラー（添え字ズレによるバグ）を起こしにくくする工夫を取り入れます。

1.3.1　レベル 2 で扱えない問題

これまでの二分探索では、f の値が切り替わる場所が少なくとも 1 つ $[l, r]$ に存在するという仮定を行っていました。しかし、実際の問題では「全域が真」「全域が偽」と

いうケースも頻繁に登場します。例えば次の問題を考えてみます。

.....

昇順ソート済みの長さ N の整数配列 A と整数 K が与えられる。A に含まれる K 以上最小の値は何かを答えよ。存在しない場合はその旨を報告せよ。

.....

例えば A の最大値が K 未満であった場合、求める値は存在しません。この問題をレベル 2 の二分探索で扱うには、

1. 二分探索を適用する前に A の最大値を確認する。
2. $A[N] = \infty$ というように、$N + 1$ 個目要素を付け足す。

というどちらかの対策を使うのが一般的です。しかし、本質的なアルゴリズムが変化していないのにかかわらず、適用する前にひと手間が必要というのは、二分探索アルゴリズムの一般化が足りていないと言えるでしょう。

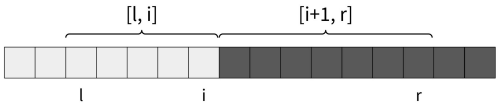
また、競技プログラミングにおいて、二分探索は別のアルゴリズムやデータ構造と併用することがかなり多いです。特に、累積和と組み合わせて使う場合などは添え字ミスをしやすくなる傾向があります。そういった場面では、できるだけ二分探索の使用前・使用後のチェックをなくしたいはずです。

1.3.2　改善案

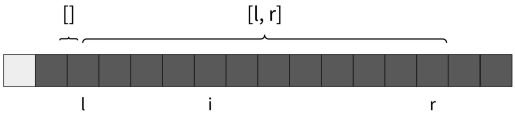
二分探索を「境界を見つけるアルゴリズム」としてとらえるのではなく、「$[l, r]$ を二つの区間に分割するアルゴリズム」としてとらえることで問題を解決することができます。具体的には閉区間 $r = [l, r]$ を受け取って、閉区間の組 (r'_1, r'_2) を返すアルゴリズムであると考えます。ただし、r'_1 に属する整数はすべて f が true になり、r'_2 に属する整数はすべて f が false になるという条件をみたします。[*3]

このようにすると、レベル 2 では対応できなかった「境界が存在しない場合」を空区間で表現できます。また、戻り値の事後条件が非常にわかりやすいため off-by-one も発生しにくいはずです。図 1.3 は区間の分割を図示したものです。戻り値を区間とすることで扱いやすくなります。

[*3] 切り替わりが複数あるような場合は成立しませんが、切り替わりをちょうど一つ見つけることは依然可能です。



▲図 1.3: レベル 2 までと同じ問題に対する区間の分割の例



▲図 1.4: 境界が存在しない問題に対する区間の分割の例

また、図 1.4 は切り替わりが存在しない場合の区間の分割の例です。このような場合も前処理や後処理を必要としません。また、誤って空区間の要素を取得しようとした場合も検出することができます。単に戻り値の整数を特別な値にするといった対応では、配列外参照などが起こるまで気が付かないかもしれません。

1.3.3　プログラム例

以上のアイデアをもとに、整数閉区間 R、関数 f を受け取り、$p \in R'_1$ なる全ての整数に対して $f(p) = \mathrm{true}$、$p \in R'_2$ なる全ての整数に対して $f(p) = \mathrm{false}$ となる二つの整数閉区間 (R'_1, R'_2) を返却する関数 `levelThree` を以下に示します。

```
struct ClosedIRange {
    private int L, R;
    private bool emp = false;

    this (int L_, int R_) {
        enforce(L_ <= R_);
        L = L_;
        R = R_;
    }
    bool empty () {
        return emp;
    }
    static ClosedIRange emptyRange () {
        auto ret = ClosedIRange();
        ret.emp = true;
```

```
        return ret;
    }
    int front () {
        enforce(!empty());
        return L;
    }
    int back () {
        enforce(!empty());
        return R;
    }
}

ClosedIRange[2] levelThree (ClosedIRange R, bool delegate(int) ›
›f) {
    assert(!R.empty());
    int l = R.front();
    int r = R.back();

    if (f(l) == f(r)) {
        if (f(l)) {
            return [
                ClosedIRange(l, r),
                ClosedIRange.emptyRange()
            ];
        }
        return [
            ClosedIRange.emptyRange(),
            ClosedIRange(l, r)
        ];
    }

    while (1 < r - l) {
        int m = (l + r) / 2;
        if (f(m)) {
            l = m;
        }
        else {
            r = m;
        }
    }
```

```
    return [
        ClosedIRange(R.front(), l),
        ClosedIRange(r, R.back())
    ];
}
```

1.4 参考文献

1. めぐる式二分探索亜種を使ってみませんか？、InTheBloom 著、https://inthebloom.github.io/post/easy-to-use-binarysearch/、2025 年 11 月 12 日閲覧
   - 自著です。レベル 2 の二分探索の問題点を指摘しています。
2. 抽象化ライブラリの第一歩としての二分探索、rsk0315 著、https://rsk0315.hatenablog.com/entry/2024/08/04/185219、2025 年 11 月 12 日閲覧
   - 本書とは少し異なる思想で二分探索を抽象化しています。
3. 因幡めぐる@競技プログラミング著、https://x.com/meguru_comp/status/697008509376835584、2025 年 11 月 12 日閲覧
   - 日本国内の競技プログラミングコミュニティで広まっている「めぐる式二分探索」と呼ばれる二分探索の実装です。本書におけるレベル 2 相当の実装だと考えられます。

第 2 章

等比数列の和の計算

定数 α, r と漸化式

$$a_1 = \alpha$$
$$a_{n+1} = ra_n$$

によって定まる数列 a を**等比数列**と言います。この章では、等比数列の先頭 n 項の和 $S_n = a_1 + a_2 + \cdots + a_n$ を高速に計算するアルゴリズムを紹介します。

2.1 再帰的解法 1

本節では S_n の構造に着目した再帰的な解法を紹介します。まず簡単のため、等比数列の初項が 1、つまり $\alpha = 1$ である場合のみを考えることにします。ただし、$\alpha = 1$ であるときの解を求めた後に全体を α 倍すれば元の問題が解けることに留意してください。以降の節においても $\alpha = 1$ のケースのみを考えます。

2.1.1 アイデア

S_n のある連続部分を切り取ったとしても、r がひとつずつ掛けられながら足されるという構造は変わりません。x における S_x がわかれば、より大きな y に対する S_y を求めるのに役立ちそうです。そこで、問題を半分に分けることを考えてみます。

n が偶数のとき、

$$\begin{aligned} S_n &= 1 + r + \cdots + r^{\frac{n}{2}-1} + r^{\frac{n}{2}} + \cdots + r^{n-1} \\ &= S_{\frac{n}{2}} + r^{\frac{n}{2}} S_{\frac{n}{2}} \end{aligned}$$

が成立します。この事実に従って適切に計算することで、問題のサイズを半分にして

いくことができます。n が奇数のときは $S_n = 1 + rS_{n-1}$ を利用して偶数のケースに帰着させてしまいましょう。

2.1.2　アルゴリズム

与えられた自然数 n に対して、(r^n, S_n) を計算する関数 f を設計します。以下、$f(n)[0], f(n)[1]$ はそれぞれ r^n, S_n を指すとします。

---

1. $n = 0$ のとき
$(1, 0)$ を返却。

2. $n = 2k$ のとき
$(f(k)[0]^2, f(k)[1] + f(k)[0] \times f(k)[1])$ を返却。

3. $n = 2k + 1$ のとき
$(r \times f(2k)[0], 1 + r \times f(2k)[1])$ を返却。

それぞれの段階は定数回の和と積を計算するだけなので $O(1)$ 回の演算を行います。再帰 2 回につき n は半分以下になるため $\Theta(\log n)$ 回の再帰で $n = 1$ に到達します。よって、このアルゴリズムは $\Theta(\log n)$ 回の演算で終了します。なお、ここで演算回数に着目したのは、多倍長整数などを用いた際に演算のコストが $O(1)$ 時間ではなくなるためです。

2.1.3　プログラム例

通常の整数の範囲で計算するために mod で割ったあまりを求めるプログラムになっています。多倍長整数を用いる場合は剰余演算を消してください。

```
long[2] f (long r, long n, const long mod) {
    if (n == 0) {
        return [1 % mod, 0];
    }
    if (n % 2 == 1) {
        auto ret = f(r, n - 1, mod);
        return [r * ret[0] % mod, (1 + r * ret[1]) % mod];
```

```
    }
    auto ret = f(r, n / 2, mod);
    return [ret[0] * ret[0] % mod, (ret[1] + ret[0] * ret[1] % ›
›mod) % mod];
}
```

2.2 再帰的解法 2

本節では、解法 1 と少し異なるアプローチで問題サイズを小さくする再帰的な解法を紹介します。

2.2.1 アイデア

$S_n = 1 + r + r^2 + r^3 + \cdots + r^{n-1}$ ですが、これを指数が偶数のものと奇数のものに分けてみます。すると、n が偶数のとき、

$$S_n = (1 + r^2 + \cdots + r^{n-2}) + (r + r^3 + \cdots + r^{n-1})$$

という $n/2$ 項の塊二つに分解することができます。そして、前者を X、後者を Y としたとき $Y = rX$ の関係が成立しています。すなわち、X の値がわかれば $S_n = (1 + r)X$ であるため元の問題が解けます。

ここで X を観察してみると、これは公比が r^2、項数が $n/2$ の等比数列の和であることがわかります。

以上より、元の問題から X を求める問題、つまり公比二乗、数列の長さが半分の問題に帰着できました。n が奇数のときは再帰的解法 1 で紹介したようにして偶数のケースに帰着させればよいです。

2.2.2 アルゴリズム

公比 r、長さ n の問題に対する解を計算する関数 $g(r, n)$ を設計します。

1. $n = 0$ のとき
0 を返却。

2. $n = 2k$ のとき
$(1 + r)g(r^2, n/2)$ を返却。

3. $n = 2k + 1$ のとき
$1 + rg(r, n - 1)$ を返却。

こちらも各段階では演算 $O(1)$ 回、再帰 2 回につき n は半分以下になるため $\Theta(\log n)$ 回の再帰で終了します。よって全体の演算回数は $\Theta(\log n)$ 回です。

2.2.3　プログラム例

```
long g (long r, long n, const long mod) {
    if (n == 0) {
        return 0;
    }
    if (n % 2 == 1) {
        return (1 + r * g(r, n - 1, mod)) % mod;
    }
    return (1 + r) * g(r * r % mod, n / 2, mod) % mod;
}
```

2.3　閉じた式を用いた解法

本節では、S_n の閉じた式による表示から計算する解法を紹介します。

2.3.1　アイデア

S_n は S_{n+1} との差を考えることで閉じた式を得ることができます。

$$\begin{aligned} S_{n+1} - S_n &= (1 + r + \cdots + r^n) - (1 + r + \cdots + r^{n-1}) \\ &= r^n \end{aligned}$$

ここで、$S_{n+1} = 1 + rS_n$ を利用すると、

$$\begin{aligned} 1 + rS_n - S_n &= r^n \\ (r-1)S_n &= r^n - 1 \\ S_n &= \frac{r^n - 1}{r-1} \end{aligned}$$

となり、閉じた式を得ることができました。

この式には r^n が含まれていますが、これは**繰り返し二乗法**というアルゴリズムを利用することで $\Theta(\log n)$ 回の演算で計算することができます。繰り返し二乗法については「2.5.1 繰り返し二乗法について」(p.18) を参照してください。

この解法には 2 つの注意点があります。

注意点 1
$r - 1 = 0$ のときは両辺を $r - 1$ で割る変形を行うことができませんが、そのような場合は $S_n = n$ となるので適切に場合分けを行えばよいです。

注意点 2
「S_n を M で割ったあまり」を求めたい場面ではこのまま適用することはできません。これは法 M において $r - 1$ が乗法逆元を持たない場合があるからです。砕けた言い方をすると、M で割ったあまりとして数を扱うとき「割り算」ができない場合があるということです。

そのような場合、次のように計算をします。

1. $r^n - 1$ を $M(r-1)$ で割ったあまりを計算する
2. その値を $r - 1$ で割る

これが正しく動作することを確認しておきましょう。$r^n - 1$ を $M(r-1)$ で割ったあまりが rem であったとします。ただし、$0 \leq \mathrm{rem} < M(r-1)$ です。この時、ある整数 k がただ一つ存在して、

$$r^n - 1 = kM(r-1) + \mathrm{rem}$$

と表示できます。S_n は常に整数となることから、$r^n - 1$ は $r - 1$ で割り切れることがわかります。ここで、$kM(r-1)$ は明らかに $r - 1$ で割り切れることから、rem も

$r-1$ で割り切ることができます。両辺を $r-1$ で割ると、

$$S_n = kM + \frac{\mathrm{rem}}{r-1}$$

となります。仮定より、$0 \leq \mathrm{rem}/(r-1) < M$ です。M で割った余りの定義を考えると、$\mathrm{rem}/(r-1)$ こそが求める値であることがわかりました。

2.3.2　アルゴリズム

S_n を計算する関数 h を設計します。

---

1. $r = 1$ のとき
n を返却。

2. $r \neq 1$ のとき
$r^n - 1$ を繰り返し二乗法で計算し、$r-1$ で割った商を返却。ただし M で割ったあまりを計算するときは $r^n - 1$ を $M(r-1)$ で割ったあまりを求める必要がある。

計算量は繰り返し二乗法が支配的になり、全体 $\Theta(\log n)$ 回の演算が必要です。

2.3.3　プログラム例

繰り返し二乗法による冪乗計算を行う関数 pow の実装は省略しています。また、mod で割ったあまりを求めるプログラムになっています。

```
long h (long r, long n, const long mod) {
    if (r == 1) {
        return n % mod;
    }
    long ret = pow(r, n, mod * (r - 1)) - 1;
    if (ret < 0) {
        ret += mod * (r - 1);
    }
    ret /= r - 1;
```

```
    return ret;
}
```

2.4　線形漸化式の行列表示による解法

本節では S_n と S_{n+1} が線形漸化式により結び付けられることを利用した解法を紹介します。

2.4.1　アイデア

S_{n+1} と S_n の間には

$$S_{n+1} = rS_n + 1$$

という関係が成立します。この関係は行列積で表現することができて、

$$\begin{pmatrix} S_{n+1} \\ 1 \end{pmatrix} = \begin{pmatrix} r & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} S_n \\ 1 \end{pmatrix}$$

となります。$S_0 = 0$ とする[*1]と、

$$\begin{pmatrix} S_n \\ 1 \end{pmatrix} = \begin{pmatrix} r & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} S_{n-1} \\ 1 \end{pmatrix}$$
$$\begin{pmatrix} S_n \\ 1 \end{pmatrix} = \begin{pmatrix} r & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} r & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} S_{n-2} \\ 1 \end{pmatrix}$$
$$\cdots$$
$$\begin{pmatrix} S_n \\ 1 \end{pmatrix} = \begin{pmatrix} r & 1 \\ 0 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

となります。あとは行列の n 乗を計算できれば解くことができます。そして、行列の n 乗も繰り返し二乗法により計算可能です。繰り返し二乗法については「2.5.1 繰り返し二乗法について」(p.18) を参照してください。

[*1] $S_0 = 0$ はここで定義しましたが、漸化式は依然成立します。

2.4.2　アルゴリズム

$\left(\begin{smallmatrix} r & 1 \\ 0 & 1 \end{smallmatrix}\right)^n$ を繰り返し二乗法で計算し、$\left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$ との積を計算して返却。

こちらも繰り返し二乗法が支配的で、行列積 1 回の計算回数を c として、$\Theta(c \log n)$ 回の演算が必要です。

2.4.3　プログラム例

繰り返し二乗法による冪乗計算を行う関数 pow の実装は省略しています。また、mod で割ったあまりを求めるプログラムになっています。

```
long i (long r, long n, const long mod) {
    auto ret = pow([r, 1, 0, 1], n, mod);
    return ret[1];
}
```

2.5　補足

2.5.1　繰り返し二乗法について

繰り返し二乗法は冪乗計算を高速に行うアルゴリズムです。例として 5^{10} を考えてみましょう。

中心的なアイデアは、$5^{10} = 5 \times 5 \times \cdots \times 5$ として 9 回の掛け算を行うのではなく、$5^{10} = 5^5 \times 5^5$ といった風に $n/2$ 乗の 2 乗として計算することです。

こうすることで、計算する必要のある指数の大きさをどんどん半分にすることができ、掛け算の回数を $\Theta(\log n)$ 回にすることができます。再帰の各段階では掛け算 1 回を行うのみなので、計算回数は $O(1)$ 回です。よって全体の計算回数も $\Theta(\log n)$ 回になります。

n が奇数のときは、$a^n = a^{n-1} \times a$ として偶数のケースに帰着させるか、$a^n =$

$a^{\lfloor n/2 \rfloor} \times a^{\lfloor n/2 \rfloor} \times a$ とすればよいです。

以下に整数 a を n 乗した値を mod で割ったあまりを計算する関数 pow を掲載します。

```
long pow (long a, long n, const long mod) {
    if (n == 0) {
        return 1 % mod;
    }
    if (n % 2 == 1) {
        return a * pow(a, n - 1, mod) % mod;
    }
    auto half = pow(a, n / 2, mod);
    return half * half % mod;
}
```

積に結合法則が成立すれば整数の冪乗以外にも利用できます。例えば 2 × 2 行列の冪乗を計算する関数は以下のようになります。

```
long[4] pow (long[4] a, long n, const long mod) {
    if (n == 0) {
        return [1 % mod, 0, 0, 1 % mod];
    }
    if (n % 2 == 1) {
        auto sm = pow(a, n - 1, mod);
        return [
            (a[0] * sm[0] + a[1] * sm[2]) % mod,
            (a[0] * sm[1] + a[1] * sm[3]) % mod,
            (a[2] * sm[0] + a[3] * sm[2]) % mod,
            (a[2] * sm[1] + a[3] * sm[3]) % mod
        ];
    }
    auto half = pow(a, n / 2, mod);
    return [
        (half[0] * half[0] + half[1] * half[2]) % mod,
        (half[0] * half[1] + half[1] * half[3]) % mod,
        (half[2] * half[0] + half[3] * half[2]) % mod,
        (half[2] * half[1] + half[3] * half[3]) % mod
    ];
}
```

2.6　参考文献

- E - Geometric Progression 解説、shakayami 著、https://atcoder.jp/contests/abc293/editorial/5972、2025 年 9 月 23 日閲覧
  - 再帰的解法 2 が紹介されています。
- E - Geometric Progression 解説、kyopro_friends 著、https://atcoder.jp/contests/abc293/editorial/5966、2025 年 9 月 23 日閲覧
  - 閉じた式を用いた解法において乗法逆元が存在しない場合の回避策が紹介されています。
- E - Geometric Progression 解説、cn449 著、https://atcoder.jp/contests/abc293/editorial/5955、2025 年 9 月 23 日閲覧
  - 線形漸化式の行列表示による解法が紹介されています。
- 繰り返し二乗法に関する頭の整理 - どこにでもいる SE の備忘録、nogawanogawa 著、https://www.nogawanogawa.work/entry/a%5En、2025 年 9 月 23 日閲覧
- 行列累乗まとめ (競プロ)、shibak3n 著、https://zenn.dev/shibak3n/articles/f08a8ad67a7d14、2025 年 9 月 23 日閲覧
  - 繰り返し二乗法の応用が紹介されています。

第 3 章

順序統計量の選択

N 要素の集合における i 番目の**順序統計量**とは、その集合における i 番目に小さな値のことを指します。特に、$i = 1$ のケースを最小値、$i = N$ のケースを最大値といいます。

集合から i 番目の順序統計量を選択するという問題は**選択問題**と呼ばれています。最も単純な解法の一つは集合をソートすることで、この場合の時間計算量は $O(N \log N)$ 時間です。しかし、実は全体をソートすることなく選択問題を解く賢いアルゴリズムが存在し、$\Theta(N)$ 時間で動作します。

本章ではこの線形時間選択アルゴリズムの動作原理とその計算量評価について説明します。

3.1 randomSelect - 期待線形時間アルゴリズム

本節では、ランダム性により期待 $\Theta(N)$ 時間を達成する選択アルゴリズム `randomSelect` を紹介します。以下、説明において N 要素の集合 S から $1 \leq i \leq N$ 番目の順序統計量を選択するものとし、集合は配列として与えられることにします。

3.1.1 アルゴリズム

まず与えられた集合から一様ランダムに要素をひとつ選択し、それをピボットとします。このピボットを用いて集合を次の 3 つに分割します。

- ピボット未満の要素の集合 S_{le}
- ピボットと等しい要素の集合 S_{eq}
- ピボット超過の要素の集合 S_{gr}

「分割」というのは、配列を先頭から「ピボット未満の値たち」、「ピボットと等しい

値たち」、「ピボット超過の値たち」という順になるように並べ替えることを指します。ピボット未満の部分とピボット超過の部分は、ピボットとの大小関係さえ守られていればよく、そこに属する要素たち自身が昇順に並んでいる必要はないことに注意してください。このような並べ替えは $|S|$ 時間で可能です。詳しいアルゴリズムは「3.1.3 プログラム例」(p.27) を参照してください。[*1]

$i \leq |S_{\mathrm{le}}|$ である場合、S_{le} の中に目的の値が存在します。よって、S_{le} に対して再帰的に `randomSelect` を適用します。$|S_{\mathrm{le}}| + |S_{\mathrm{eq}}| < i$ である場合も同様で、S_{gr} にに対して再帰的に `randomSelect` を適用します。この場合、目的の順序統計量は $i - |S_{\mathrm{le}}| - |S_{\mathrm{eq}}|$ 番目にずれることに注意が必要です。それ以外の場合は S_{eq} の要素の一つが解です。

このアルゴリズムはクイックソートと似た原理で動きますが、分割した 3 つの集合のうち解が存在しないものは放置するという差があります。したがって、アルゴリズムの性能はピボットを適切に選べるかに依存します。

良いケースとして、ピボットとして常に中央値を取り続ける場合を考えてみましょう。このとき、分割 1 回につき考えるべき集合のサイズは半分以下になります。よって、時間計算量は $N + N/2 + N/4 + \cdots = N(1 + 1/2 + 1/4 + \ldots) \in \Theta(N)$ で抑えられます。

一方、悪いケースとして、ピボットとして常に最大値を取ってしまう場合を考えてみましょう。このとき、分割 1 回につき取り除けるのは 1 要素のみとなり、この時の時間計算量は最悪時で $N + N - 1 + N - 2 + \cdots + 1 \in \Theta(N^2)$ となってしまいます。

しかし、ありえる操作列全体から見たとき、良いケースの割合は悪いケースより十分に多いです。結果として、期待計算量は $\Theta(N)$ となることが示せます。次の節ではその解析を行います。

3.1.2　計算量の解析

解析のために、用語を導入します。考えている集合の要素数を N としたとき、昇順で先頭 $\lfloor N/4 \rfloor$ 個に来る値からなる集合を**前半部分**、後ろ $\lfloor N/4 \rfloor$ に来る値からなる集合を**後半部分**、残りを**中間部分**と呼びます。これらは元の S における位置ではなく、「昇順に並べたとしたら先頭（後ろ）$\lfloor N/4 \rfloor$ 個に来る値の集合」を考えていることに注意してください。

[*1] クイックソートの `partition` 関数とほぼ同様のアルゴリズムを用います。

また、考える対象の集合の要素数が分割前の集合の要素数の $3/4$ 倍以下になるような分割を**良い分割**、そうでない分割を**悪い分割**と呼びます。

解析のために 3 つの補題を証明します。

補題 1
ピボットとして中間部分に属する要素が選ばれた際、必ず良い分割となる。

証明
選択したい要素が何番目かにかかわらず、前半部分と後半部分の少なくとも片方は分割後無視できます。また、ピボットそのものも無視できるため、分割によって集合の要素数は $\lfloor N/4 \rfloor + 1$ 以上減ります。$N = 4k + r\ (0 \leq r < 4)$ と表示できることを利用すると、

$$\begin{aligned} N - \left\lfloor \frac{N}{4} \right\rfloor - 1 &= 4k + r - k - 1 \\ &= 3k + r - 1 \\ &= \left(3k + \frac{3}{4}r\right) + \left(\frac{1}{4}r - 1\right) \\ &\leq 3k + \frac{3}{4}r \\ &= \frac{3}{4}N \end{aligned}$$

が成立するため、分割が良い分割であることが示されました。（証明終わり）

補題 2
ピボットとして中間部分に属する要素を選ぶ確率は $1/2$ 以上である。

証明
ピボットは一様ランダムに選択するため、中間部分に属する要素の数が全体の $1/2$ 以上であればよいです。補題 1 と同様に $N = 4k + r\ (0 \leq r < 4)$ と表示すると、

$$\begin{aligned} N - 2\left\lfloor \frac{N}{4} \right\rfloor &= 4k + r - 2k \\ &= 2k + r \\ &\geq 2k + \frac{r}{2} \\ &= \frac{N}{2} \end{aligned}$$

以上より示されました。（証明終わり）

補題 3

確率 $0 < p \leq 1$ で成功する試行を成功するまで繰り返すとき、試行回数の期待値は $1/p$ である。ただし、それぞれの試行は独立であるとする。

証明

k 回目の試行で初めて成功する確率は $(1-p)^{k-1}p$ であるため、試行回数の期待値を E とすると、

$$E = \sum_{k=1}^{\infty} k(1-p)^{k-1}p$$

となります。この無限和を求めるために、まず先頭 n 項の有限和

$$E_n = \sum_{k=1}^{n} k(1-p)^{k-1}p$$

を考えます。この値は $(1-p)E_n$ との差を考えると計算できて、

$$\begin{aligned}
E_n - (1-p)E_n &= p\sum_{k=1}^{n}\{(1-p)^{k-1}(k-(k-1))\} - n(1-p)^n p \\
pE_n &= p\sum_{k=0}^{n-1}\{(1-p)^k\} - n(1-p)^n p \\
&= p\frac{(1-p)^n - 1}{(1-p) - 1} - n(1-p)^n p \\
&= 1 - (1-p)^n - n(1-p)^n p \\
E_n &= \frac{1}{p} - (1-p)^n p - n(1-p)^n
\end{aligned}$$

となります。仮定より $0 \leq 1-p < 1$ であるため、

$$\begin{aligned}
E &= \lim_{n\to\infty} E_n \\
&= \frac{1}{p}
\end{aligned}$$

より $E = 1/p$ がわかります。（証明終わり）

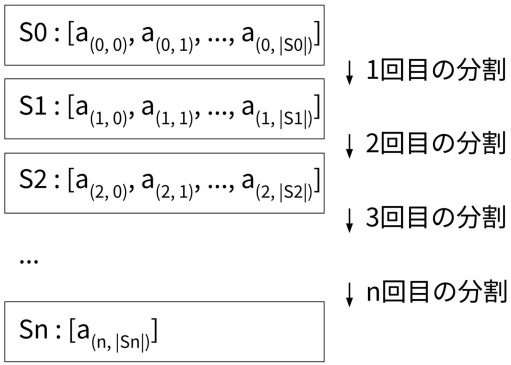
必要な補題が準備できたので、ここから証明していきましょう。

命題
randomSelect は期待 $\Theta(N)$ 時間で動作する。

証明
「i 番目の順序統計量が見つかった時点で終了」ではなく、「集合の要素数が 1 になった時点で終了」というより強い条件で考えます。また、S_x を x 回目の分割が終わった後の集合とします。ただし、初期状態は 0 回目の分割が終わった後とみなし、S_0 とします。

randomSelect におけるある一連の試行に対して、分割が何回発生したかを表す確率変数 n と良い分割が何回発生したかを表す確率変数 m を定めます。また、i 番目に発生した良い分割は何番目分割であったかを表す確率変数 h_i を定めます。後の解析を簡単にするため、0 回目の分割と m 回目の分割は良い分割であったとします。つまり、$h_1 = 0, h_m = n$ です。

これらにより、i 回目の良い分割を引いてから $i+1$ 回目の良い分割を引くまでに何回分割したかを表す確率変数 $X_i = h_{i+1} - h_i$ を定義できます。



▲ 図 3.1: 確率変数の図示

図図 3.1 は定義した確率変数 n および各段階の列 S_x を図示したものです。

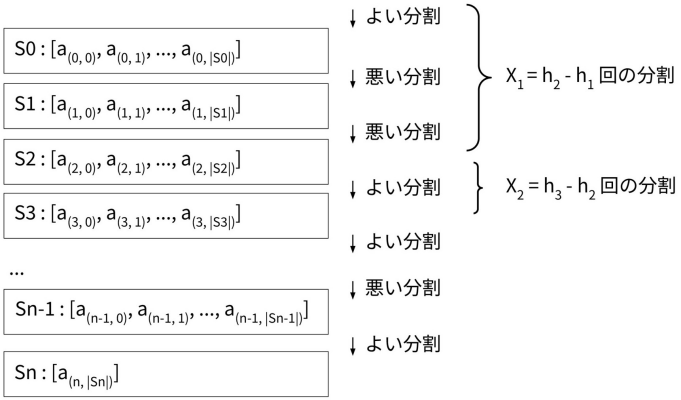
細かい準備ができました。randomSelect における計算量を評価していきましょう。randomSelect における計算は、

1. $O(1)$ 時間でピボットを選択する
2. $\Theta(|S|)$ 時間かけて分割を行う
3. 再帰的に問題を解く

という 3 ステップからなり、ステップ 2 が支配的であることを踏まえると、ある試行における計算量 T は次のように評価できます。

$$
\begin{aligned}
T &= \sum_{i=0}^{n-1} |S_i| \\
&= \sum_{j=1}^{m-1} \sum_{k=h_j}^{h_{j+1}-1} |S_k| \\
&\leq \sum_{j=1}^{m-1} X_j |S_{h_j}|
\end{aligned}
$$

この変形を視覚的に説明した図が図 3.2 です。2 行目への変形は、「よい分割」から次の「よい分割」をグループとして見ています。3 行目への変形は、1 つのグループを分割回数とグループ内最大の配列長の積で上から評価しています。



▲図 3.2: 式変形の視覚的な説明

この期待値を考えます。

$$
\begin{aligned}
E[T] &\leq E\left[\sum_{j=1}^{m-1} X_j |S_{h_j}|\right] \\
&= \sum_{j=1}^{m-1} E[X_j]|S_{h_j}| \\
&\leq 2\sum_{j=1}^{m-1} |S_{h_j}| \\
&\leq 2\sum_{j=1}^{m-1} \left(\frac{3}{4}\right)^{j-1} |S_0| \\
&\leq 2|S_0| \sum_{j=0}^{\infty} \left(\frac{3}{4}\right)^{j} \\
&= 8|S_0|
\end{aligned}
$$

ただし、2 行目の変形は期待値の線形性を利用しました。3 行目の変形は補題 2 と補題 3 を利用しました。4 行目の変形は補題 1 を利用しました。仮定より、$|S_0| = N$ であるため、期待計算量は $8N$ 以下になります。ここから $O(N)$ であることが従います。

また、初回分割に必ず $\Theta(N)$ かかることから、期待計算量が $\Theta(N)$ であることが示されました。（証明終わり）

3.1.3 プログラム例

関数 randomSelect の実装例を以下に示します。この関数は配列 A と整数 $0 \leq k$ を受け取り、A の k 番目の値を返します。

```
int randomSelect (int[] A, int k) {
    import std.exception: enforce;
    import std.algorithm: sort, min, swap;
    import std.random: uniform;
    enforce(0 <= k && k < A.length);

    const int N = cast(int)(A.length);

    // ピボットの選択
    int piv = A[uniform(0, N)];
```

```
    // 集合の分割
    int loCount = 0;
    int hiCount = 0;
    foreach (i; 0 .. N) {
        if (A[i] < piv) {
            swap(A[i], A[loCount]);
            loCount++;
        }
    }
    foreach_reverse (i; 0 .. N) {
        if (piv < A[i]) {
            swap(A[i], A[N - hiCount - 1]);
            hiCount++;
        }
    }

    // 再帰的に呼び出し
    if (k + 1 <= loCount) {
        return randomSelect(A[0 .. loCount], k);
    }
    if (N - hiCount < k + 1) {
        return randomSelect(A[N - hiCount .. N], k - (N - hiCou›
›nt));
    }
    return piv;
}
```

3.2　select - 線形時間アルゴリズム

本節では、決定的に $\Theta(N)$ 時間を達成する選択アルゴリズム select を紹介します。中心的なアイデアはピボット選択を賢く行うことであり、これに由来して「中央値の中央値」という名前で呼ばれることがあります。

3.2.1　アルゴリズム

説明および正当性の証明を簡単にするため、N を 5 の倍数とします。そうでないときには $\Theta(N)$ 時間かけて 5 の倍数にすることができます。具体的には、まず $\Theta(N)$

時間かけて最小値を探し、先頭要素と swap します。次に選択したい値が最小値ならその値を返却、そうでなければ残った $N-1$ 要素に対して select を適用するといったことを高々 4 回行えばよいです。

まず与えられた配列を 5 要素ごとの小配列 $N/5$ 個としてみなし、それぞれをソートします。小配列の長さが入力の大きさに関係なく定数で定められているため、これは $\Theta(N)$ 時間で行えることに注意してください。

次にそれぞれの小配列の中央値を配列の先頭 $N/5$ 個に集め、その $N/5$ 個に対して再帰的に select を適用し、それらの中央値を求めます。この「中央値の中央値」をピボットとします。あとは randomSelect と同様に再帰的に select を適用します。詳しいアルゴリズムは「3.1.1 アルゴリズム」(p.21) を参照してください。

このアルゴリズムは以下の 2 つの難しさがあります。

- ピボットの選択にも select を利用しているため、計算量がわかりにくい
- 小配列の中央値たちの中央値が本当に十分良いピボットなのかが自明でない

さあ、アルゴリズムの解析をしていきましょう。

3.2.2 計算量の解析

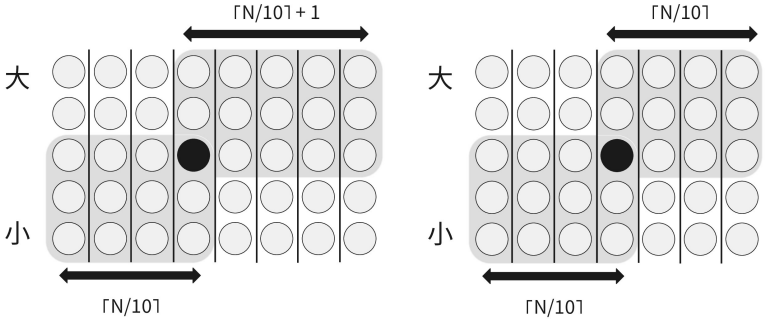
まずは、ピボットの性能に関する補題を証明しましょう。

補題 1
小配列の中央値たちの中央値をピボットとして利用したとき、分割によって集合の要素数が $7/10$ 倍以下になる。

証明
集合の要素数を N とします。「3.2.1 アルゴリズム」(p.28) で紹介した方法を用いて N を 5 の倍数になるようにしておきます。

図 3.3 は $N/5$ が奇数、偶数それぞれの場合において中央値の中央値をピボットにした際の様子を表しています。各列は小配列を表し、下から上に昇順に並んでいます。各列の 3 行目は小配列の中央値を表しており、黒く塗りつぶされた丸はそれらの中央値、つまりピボットを表します。図の四角形で囲まれた領域は、ピボット以下、またはピボット以上の要素を表します。

選択したい順序統計量が小さい方から何番目であったとしても、四角で囲まれた領域



▲図 3.3: 分割によって除かれる要素の図示。黒丸は中央値の中央値を表す。

の少なくとも片方、つまり $3\lceil N/10 \rceil$ を取り除くことができます。[*2]

残る要素数は $N - 3\lceil N/10 \rceil$ 以下となります。$N/10 \leq \lceil N/10 \rceil$ なので、この差を $d = \lceil N/10 \rceil - N/10$ とするとき、

$$\begin{aligned} N - 3\left\lceil \frac{N}{10} \right\rceil &= N - 3\left(\frac{N}{10} + d\right) \\ &= \frac{7}{10}N - 3d \\ &\leq \frac{7}{10}N \end{aligned}$$

より、分割により $7/10$ 倍以下になることが示されました。（証明終わり）

次は `select` の計算量に関する漸化式を立てます。

補題 2
要素数 N の集合に対する `select` の計算量が $T(N)$ 以下となるとき、漸化式

$$T(N) \leq T\left(\frac{N}{5}\right) + T\left(\frac{7}{10}N\right) + \Theta(N)$$

が成立する。

証明

[*2] 選択したい順序統計量とピボットが等しい場合は $3\lceil N/10 \rceil$ 要素取り除けない場合がありますが、この場合は集合の分割後すぐさま答えが得らえるため考慮していません。

「3.2.1 アルゴリズム」(p.28) の動作それぞれの計算量を細かく見ていきます。

1. N を 5 の倍数にするのに $\Theta(N)$ 時間
2. 小配列のソート、それらの中央値を前に集めるのに $\Theta(N)$ 時間
3. ピボットを選ぶための再帰的な select に $T(N/5)$ 時間
4. ピボットによる配列の分割に $\Theta(N)$ 時間
5. 分割された配列に対する再帰的な select に $T(7N/10)$ 時間

がかかります。以上より、

$$T(N) \leq T\left(\frac{N}{5}\right) + T\left(\frac{7}{10}N\right) + \Theta(N)$$

が成立します。ただし、ステップ 5 の評価に補題 1 を用いました。(証明終わり)

最後に漸化式を用いて $T(N)$ を評価します。

定理
$T(N) \in \Theta(N)$ である。

証明
まずは下から抑えます。最初の再帰には必ず $\Theta(N)$ 時間かかる処理が入るため、$\Omega(N)$ 時間であることがわかります。

次に上から押さえます。ある定数 c が存在して $T(N) \leq cN$ が成立すると仮定して、置き換え法を適用します。補題 2 により、

$$\begin{aligned} T(N) &\leq \frac{1}{5}cN + \frac{7}{10}cN + \Theta(N) \\ &= \frac{9}{10}cN + \Theta(N) \end{aligned}$$

となります。$\Theta(N)$ の部分は、定義により、ある定数 d によって $\Theta(N) \leq dN$ と評価できます。[*3] これを加味して

[*3] 関数 f が $O(N)$ に属するとは、ある正定数 n_0, k が存在して、任意の $n_0 \leq n$ において $0 \leq f(n) \leq kn$ となることをいいます。今回は関数の定義域が正整数であるとして、$d = \max(f(1), f(2), \ldots, f(n_0), k)$ というようにとることができます。

$$
\begin{aligned}
T(N) &\leq \frac{9}{10}cN + dN \\
&= \left(\frac{9}{10}c + d\right) N
\end{aligned}
$$

となります。ここで、$d \leq c/10$ となるように定数 c を定めることで

$$
\begin{aligned}
T(N) &\leq \left(\frac{9}{10}c + d\right) N \\
&\leq cN
\end{aligned}
$$

が成立します。以上より、$T(N) \in O(N)$ が示されました。最初に議論した下界 $\Omega(N)$ と合わせて、$T(N) \in \Theta(N)$ がわかります。（証明終わり）

3.2.3　プログラム例

以下の実装例では N を 5 の倍数に合わせることはせず、代わりに $N = 1$ を基底ケースとして扱っています。解析は若干面倒になりますが、$\Theta(N)$ 時間であることが示せます。

```
int select (int[] A, int k) {
    import std.exception: enforce;
    import std.algorithm: sort, min, swap;
    enforce(0 <= k && k < A.length);

    const int N = cast(int)(A.length);
    immutable int blockSize = 5;

    // 基底ケース
    if (N == 1) {
        return A[0];
    }

    // 小配列のソート
    int count = 0;
    while (blockSize * count < N) {
        int lower = blockSize * count;
        int upper = min(N, lower + blockSize);
```

```
        sort(A[lower .. upper]);
        swap(A[lower], A[lower + (upper - lower) / 2]);
        count++;
    }

    // 中央値を前に詰める
    foreach (i; 0 .. count) {
        swap(A[i], A[blockSize * i]);
    }

    // ピボット選択
    int piv = select(A[0 .. count], count / 2);

    // 集合の分割
    int loCount = 0;
    int hiCount = 0;
    foreach (i; 0 .. N) {
        if (A[i] < piv) {
            swap(A[i], A[loCount]);
            loCount++;
        }
    }
    foreach_reverse (i; 0 .. N) {
        if (piv < A[i]) {
            swap(A[i], A[N - hiCount - 1]);
            hiCount++;
        }
    }

    // 再帰的に呼び出し
    if (k + 1 <= loCount) {
        return select(A[0 .. loCount], k);
    }
    if (N - hiCount < k + 1) {
        return select(A[N - hiCount .. N], k - (N - hiCount));
    }
    return piv;
}
```

3.3　補足・発展的な話題

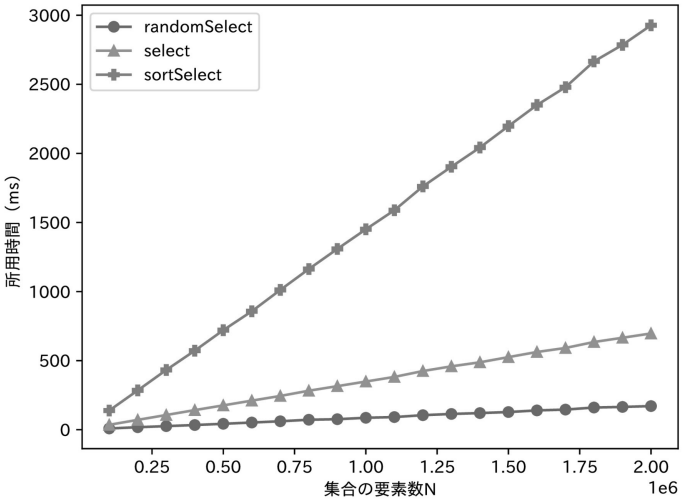
3.3.1　小配列の長さについて

select では最初に配列を長さ 5 ずつに分割しましたが、この小配列の長さは 5 でなければいけないということではありません。5 以上の奇数を選べばほとんど同様に線形時間で動作することが示せます。重要なのは分割の長さではなく、漸化式右辺に登場する T の引数の合計が 1 未満の正定数 p によって pN 以下となることです。

実際、[1] には小配列の長さを 3 に設定したうえで線形時間を達成する select の変種アルゴリズムが紹介されています。

3.3.2　選択アルゴリズムの実測値

線形時間の選択アルゴリズムが実際どれほど高速なのかを測定しました。



▲図 3.4: 各選択アルゴリズムにおける集合の要素数と所用時間の関係

図 3.4 は測定結果を matplotlib でプロットしたものです。sortSelect としてプロットされているデータは D 言語標準ライブラリの std.algorithm: sort を利用

した select で、$O(N \log N)$ 時間で動作します。同じデータに対して中央値を選択するのにかかった時間を 100 回測定し、その平均値をプロットしています。測定に用いたプログラムは https://github.com/InTheBloom/UEComic9/で公開してあります。より細かい条件はプログラムを参照してください。

図からは randomSelect、select、sortSelect の順で性能が良さそうであるということがわかります。また、2 つの線形時間選択アルゴリズムが sortSelect と比較してかなり高速に動作していることがわかります。したがって、サイズの大きな選択問題に対してある程度実用的であると言えそうです。

3.3.3　置き換え法について

select の計算量を上から評価する際に置き換え法を用いました。置き換え法とは漸化不等式を満たす関数を評価する際に用いられる解法の一つです。

適用する際の流れは、漸化式を満たす数列を一般項の推定から解く方法と似ています。例として、「3.2.2 計算量の解析」(p.29) で扱った漸化式を抑える一連の流れを追っていきましょう。漸化式を再掲しておきます。

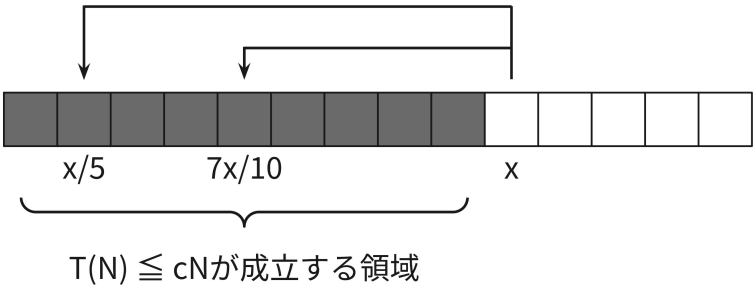
$$T(N) \leq T\left(\frac{N}{5}\right) + T\left(\frac{7}{10}N\right) + \Theta(N)$$

1. ある定数 c が存在して、$T(N) \leq cN$ が成立すると**仮定**する
2. 漸化式の右辺の T を手順 1 の仮定に基づいて置き換える
3. 右辺が cN で抑えられることを確認する

これらを確認することで、数学的帰納法のように全ての N について上から押さえることができます。

図 3.5 は置き換え法の正当性を視覚的に表現した図です。定数 c に対して任意の N で漸化式が正しいと言えた場合、十分小さい領域に対して $T(N) \leq cN$ が成立すると、それより後ろの場所でもドミノ倒しのように正しさが伝搬します。そのため、定数 c は漸化式が正しくなる程度に大きいだけではなく「十分小さい領域」で T の評価が正しくなるように取る必要があります。

置き換え法は落とし穴があります。それは漸化式の評価を**定数倍を含めて**考えなければならないという点です。特に、漸化式に漸近評価された項が存在するときは要注意です。例えば [1] で紹介されている誤りの例では



▲図 3.5: 置き換え法の視覚的な説明。未確定領域は確定領域のみに依存していることがわかる。

$$T(N) \leq 2T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + \Theta(N)$$

という漸化式を $T(N) \in O(N)$ であると推定して置き換え、

$$\begin{aligned} T(N) &\leq 2 \cdot O\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + \Theta(N) \\ &= 2 \cdot O(N) + \Theta(N) \\ &= O(N) \end{aligned}$$

というものがあります。しかしこれはビッグオーに隠された定数倍を正しく評価できておらず、「ドミノ倒し」が成立していません。明示的に $T(N) \leq cN$ と書いて置き換えを行うと、

$$\begin{aligned} T(N) &\leq 2\left(c\left\lfloor \frac{N}{2} \right\rfloor\right) + \Theta(N) \\ &\leq cN + \Theta(N) \end{aligned}$$

と評価できますが、**どのように** c を取ったとしても、$T(N) \leq cN$ を言えません。置き換え法を用いる際は漸化式が定数倍を含めて成立するように気を付ける必要があります。

3.3.4 応用

はじめは randomSelect を用いて、再帰が深くなってきたときに select に切り替えるという introSelect というアルゴリズムが存在します。このアルゴリズムは平均性能を向上させつつ必ず線形時間で動作します。

他にも、アルゴリズム中に順序統計量を定数回求める必要がある場合、このアルゴリズムによりソートの $O(N \log N)$ を回避できるため、全体の計算量から $\log N$ を取り除ける可能性があります。

また、これらのアルゴリズムを用いてクイックソートのピボットを選択すると、決定的に $O(N \log N)$ 時間で動作するクイックソートを作ることができます。ただしピボット選択が $O(1)$ 時間から $\Theta(N)$ 時間に悪化してしまうこと、また select に隠れた定数倍が大きいため、乱択のクイックソートよりも平均性能が低下すると思われます。現実的には 2 種類のソートを組み合わせるイントロソートのようなアルゴリズムを用いるのが良いでしょう。

3.4 参考文献

1. アルゴリズムイントロダクション 第 4 版 第 1 巻、T. コルメン・C. ライザーソン・R. リベスト・C. シュタイン著、浅野 哲夫・岩野 和生・梅尾 博司・小山 透・山下 雅史・和田 幸一訳、近代科学社
   - 議論の大部分の参考にしました。

第 4 章

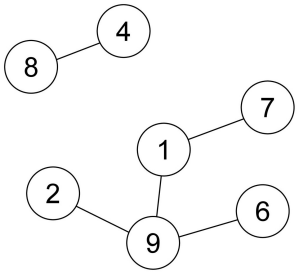
二分探索木によるデータの管理

本章では、二分探索木というデータ構造を用いて効率的にデータを管理する方法を紹介します。

4.1 グラフ

二分探索木はグラフという数理モデルの一種です。ここでいうグラフは「$y = ax$ のグラフ」といったものではなく、「モノとモノのつながり方」を数学的に記述したものです。グラフは**頂点**とそれらを結ぶ**辺**で構成されます。数学的には有限集合 V と V 上の**二項関係**[*1] E（辺、つまり「どの頂点とどの頂点が結ばれているか」という関係）の組 $G = (V, E)$ をグラフと呼びます。[*2]

図 4.1 は $V = \{1, 2, 4, 6, 7, 8, 9\}, E = \{(1, 7), (1, 9), (2, 9), (4, 8), (6, 9)\}$ なるグラフ G を図示したものです。



▲図 4.1: グラフの例

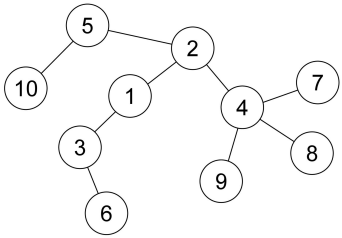
[*1] V 上の二項関係とは、直積集合 $V^2 = V \times V$ の部分集合のことをいいます。$u, v \in V$ に対して、$(u, v) \in E$ であれば辺で結ばれている、そうでなければ結ばれていないということを表現します。

[*2] 本章では**無向グラフ**を扱います。慣例に従って、$(u, v) \in E$ という表記をした際には $(v, u) \in E$ も成立することを暗黙に仮定します。また、多重辺や自己ループなども基本的には扱いません。

4.2　木/根付き木

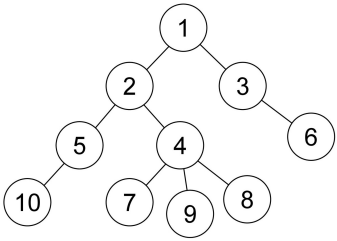
頂点集合 V が空であるグラフと、全体が**連結**であり、その辺の数 $|E|$ が頂点集合の数 $|V|$ よりも 1 小さいようなグラフを**木**といいます。図 4.2 は木の例です。

なお、グラフ全体が連結であるとは、グラフの任意の 2 頂点間を辺を通って行き来できることを指します。また、頂点 i から頂点 j へ辺をたどって移動できるとき、通った頂点列を i から j の**経路**と呼びます。



▲図 4.2: 木の例

さらに、木の頂点の 1 つを他と区別して**根**とした木を**根付き木**と言います。通常、根付き木は根を一番上にして図示されます。図 4.3 は図 4.2 の頂点 1 を根とした根付き木を図示したものです。



▲図 4.3: 図 4.2 の頂点 1 を根とした際の根付き木

根付き木の接続関係にはある種の向きが定められます。ある頂点と直接接続された頂点のうち、根との距離[*3]が最も近いものを**親**、それ以外のものを**子**と呼びます。子を

[*3] グラフにおいて、2 つの頂点の**距離**とは一方から出発してもう一方の頂点へ到達するまでに経由する辺の数の最小値を言います。連結でないペアの距離は定義されないか、∞ として扱います。頂点 x と頂点 x の距離は 0 です。また、木において、同じ頂点を 2 度通らない経路は一意です。

持たない頂点を**葉**といい、子を持つ頂点を**内点**といいます。根は親を持たない唯一の頂点です。例えば図 4.3 において頂点 2 の親は頂点 1、子は頂点 4 と頂点 5 です。

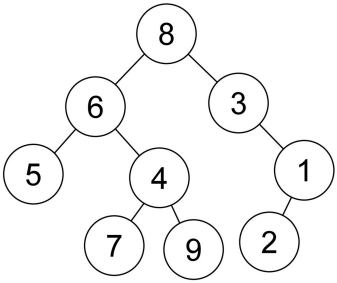
頂点の根からの距離をその頂点の**深さ**といいます。木の頂点の最大の深さを木の**高さ**と言います。図 4.3 において頂点 5 の深さは 2、木の高さは 3 です。

ある頂点の親の親の... 親という関係にある頂点をまとめて**祖先**、子の子の... 子という関係にある頂点をまとめて**子孫**と呼びます。ただし、自分自身は自分の子でも祖先でもありません。また、図 4.3 における頂点 2 と頂点 3 のようなペアは互いに子孫でも祖先でもありません。根以外の頂点から見ると、根は祖先のひとつです。

根付き木のある頂点に対して、自身とその子孫、それらを結ぶ辺のみから成る部分グラフをその頂点を根とする**部分木**と呼びます。図 4.3 において頂点 1 の部分木は木全体を指し、頂点 4 の部分木は $V = \{4, 7, 8, 9\}, E = \{(4, 7), (4, 8), (4, 9)\}$ です。

4.3　二分木

全ての頂点において子の数が 2 以下であり、それらが**左子**、**右子**として区別される根付き木を**二分木**といいます。通常、図において左子は左下、右子は右下に描かれます。図 4.4 は二分木の例です。



▲図 4.4: 二分木の例

左子、右子という区別は子を 1 つしか持たない場合でも有効であることに注意してください。例えば図 4.4 において頂点 1 は左子のみを持ち、右子を持ちません。

また、より形式的には次のように定義されます。

- 空木（頂点を持たない根付き木）は二分木である。

- 頂点 r と二分木 t_1, t_2 があるとき、r の左子を t_1、r の右子を t_2 とし、頂点 r を根とする根付き木は二分木である。ただし t_1 や t_2 が空木である場合は対応する子が存在しないものとする。
- これら 2 つの規則を有限回繰り返して得られるものだけが二分木である。

4.4　二分探索木

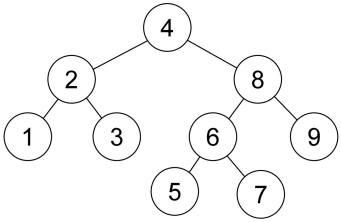
各頂点にデータが 1 つずつ書き込まれた値付きの二分木を考えます。ただし、データ同士は適切な大小関係が定義できるものとします。本書では、簡単のためデータは整数値を取るとし、通常の大小関係 $\leq$ を考えます。

このような二分木であって、全ての頂点が次の**二分探索木条件**を満たすものを**二分探索木**といいます。

- 左部分木（左子の部分木）の任意の頂点に書き込まれたデータが自身に書き込まれたデータ以下である。
- 右部分木の任意の頂点に書き込まれたデータが自身に書き込まれたデータ以上である。

ただし、以降の章では、特に断りが無ければデータは重複しないものとします。この条件を明示したのは、二分探索木に保持するデータの重複を許す場合と許さない場合でアルゴリズムに若干差異が出るためです。

また、二分探索木では各頂点に書き込まれたデータが重要であり、その頂点番号はあまり重要ではありません。そこで、以後の図では特に断りが無い場合、頂点番号**ではなく**、その頂点に書き込まれたデータを表示した図を示します。



▲図 4.5: 二分探索木の例

図 4.5 は二分探索木の例です。前述の通り書き込まれたデータのみを表示していま

す。また、二分探索木条件を満たしていることが確認できます。

4.4.1　プログラム上での表現

二分探索木をプログラム上で扱うときは、「左子と右子へのポインタ」と「自身の持つデータ」を構造体にまとめて表現します。

```
struct Node {
    Node* lch = null;
    Node* rch = null;
    int val;
}
```

実際に利用する際は、木の根へのポインタだけ保持します。以降の節で紹介するプログラムは上記の構造体 Node を利用します。

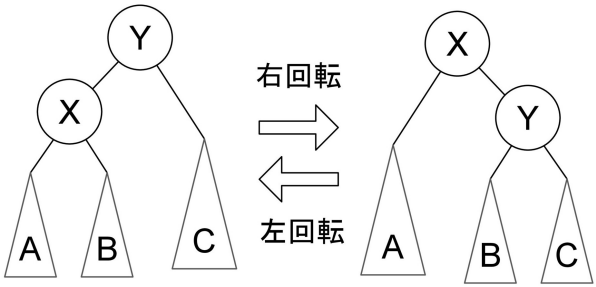
例えば図 4.5 と同じ構造の二分探索木を手動で組み立てると次のようになります。

```
Node* root = new Node(null, null, 4);
root.lch = new Node(null, null, 2);
root.lch.lch = new Node(null, null, 1);
root.lch.rch = new Node(null, null, 3);
root.rch = new Node(null, null, 8);
root.rch.rch = new Node(null, null, 9);
root.rch.lch = new Node(null, null, 6);
root.rch.lch.lch = new Node(null, null, 5);
root.rch.lch.rch = new Node(null, null, 7);
```

4.4.2　二分探索木の回転

二分探索木条件を保ったまま局所的に木の形を変える**回転**という操作があります。

図 4.6 は回転を図示したものです。図の三角形は部分木を表しており、空木も許容されます。回転は右回転と左回転があり、左子を持たない場合は左回転を行うことができず、右子を持たない場合は右回転を行うことができません。本書では回転を頂点に対する操作とみなし、「左の木の頂点 Y に右回転を行うと右の木になり、右の木の頂点 X に左回転を行うと左の木になる」と表現します。



▲図 4.6: 二分探索木の回転。左から右へ変形する操作が右回転、右から左へと変形する操作が左回転と呼ばれる。

回転により木の形状は変化しますが、どの頂点においても二分探索木条件は保たれたままです。簡単のため頂点 X, Y の親が無い場合を図示しましたが、親があったとしてもやはり二分探索木条件は保たれることに注意してください。

回転は直接接続されたある 1 つの頂点との上下関係を逆転させる操作であるとみなすことができます。回転を繰り返すことで特定の頂点を根まで移動させたり、逆に葉まで移動させたりすることができます。ある頂点の位置を狙った位置に移動させることや、木全体の形状を整えるために利用されることが多いです。

以下にプログラム例を示します。引数で指定された頂点に対して回転を行い、回転を適用した頂点がもともとあった場所に来る頂点を返却しています。

```
Node* rotL (Node* cur) {
    Node* rch = cur.rch;
    cur.rch = rch.lch;
    rch.lch = cur;
    return rch;
}

Node* rotR (Node* cur) {
    Node* lch = cur.lch;
    cur.lch = lch.rch;
    lch.rch = cur;
    return lch;
}
```

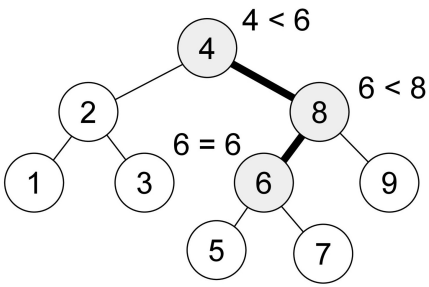
4.5　二分探索木上のデータ操作

データをわざわざ二分探索木という複雑な構造に乗せたのは、配列上で管理するよりも便利な操作を高速に行うことができるからです。本節では二分探索木に保持したデータに行うことができる操作とその時間計算量を紹介します。

4.5.1　find - 要素の検索

find はデータの中にある値が含まれるかを判定する操作です。二分探索木条件のおかげで非常に簡単に探索できます。

検索対象の値を x としましょう。まず最初に根を見ます。根のデータが x より大きければ x は（含まれるなら）左部分木に含まれることがわかります。x より小さければ右部分木に含まれることがわかります。x が含まれる方の子に進み、同様のことを x と等しいデータを持つ頂点にたどり着くまで続けます。



▲図 4.7: find(6) の動作の様子

図 4.7 は find(6) の動作を示した図です。辺をたどって 6 のデータを持つ頂点を発見するまで「潜って」行く様子がわかります。

目的のデータが見つかるか、空木にたどり着くまで子へと進み続けます。そのため計算量は木の高さを h として $O(h)$ 時間です。

以下に find のプログラム例を示します。再帰関数を利用するとシンプルになります。

```
Node* find (Node* cur, int x) {
    if (cur is null) {
```

```
        return null;
    }
    if (x < cur.val) {
        return find(cur.lch, x);
    }
    if (cur.val < x) {
        return find(cur.rch, x);
    }
    return cur;
}
```

この関数は cur の部分木に含まれる x をデータに持つ頂点へのポインタ、または null を返します。

4.5.2　pred/succ - 直近要素の検索

find とほぼ同じアルゴリズムでもう少し複雑なことが行えます。pred(x) は x 以下最大のデータを検索し、succ(x) は x 以上最小のデータを検索します。[*4]

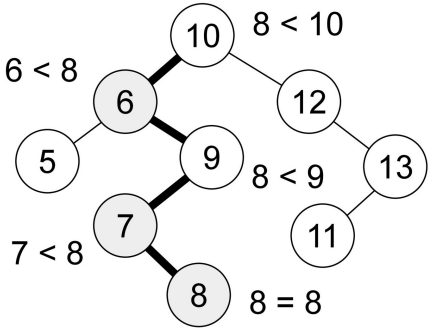
両者は条件などが反転する程度の差しかないため、ここでは pred の解説のみ行います。まずはもう少し細かく仕様を定めます。pred(cur, x) を頂点 cur の部分木における x 以下最大のデータをもつ頂点のポインタを返却する関数としましょう。

まず、cur の部分木が空木、つまり cur が null であるときは null を返却します。そうでなく、cur の持つデータが x 超過なら求める頂点は左部分木に存在するので pred(cur.lch, x) を返却します。

これらに該当しない場合、まず cur の持つデータは解の候補の 1 つです。しかし、さらに大きな値（x 以下）を持つ頂点が cur の**右部分木**に含まれている可能性があります。そこで、cur の右部分木を再帰的に探索します。もしそこでより良い解が見つかればそれを採用し、見つからなければ cur を解とします。

図 4.8 は pred(8) の動作を図示したものです。灰色の頂点はそのデータが 8 以下の頂点で、これらが解の候補になります。このうち最も 8 に近い値をもつ頂点が返却されます。アルゴリズムがデータが 8 以下の頂点と 8 超過の頂点の狭間を反復するように探索しているのがわかります。

[*4] pred、succ はそれぞれ predecessor、successor の略です。



▲図 4.8: pred(8) の動作の様子。灰色の頂点は解の候補。

この操作は空木にたどり着くまで木を降りて行くだけなので、計算量は木の高さ h として $O(h)$ 時間です。

以下に pred と succ のプログラム例を示します。

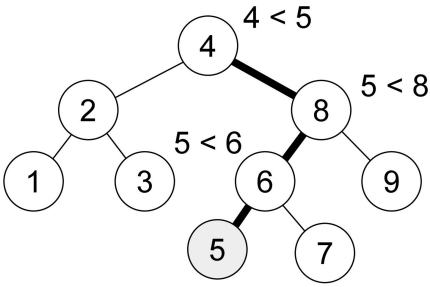
```
Node* pred (Node* cur, int x) {
    if (cur is null) {
        return null;
    }
    if (cur.val <= x) {
        Node* nex = pred(cur.rch, x);
        if (nex is null || nex.val < cur.val) {
            return cur;
        }
        return nex;
    }
    return pred(cur.lch, x);
}

Node* succ (Node* cur, int x) {
    if (cur is null) {
        return null;
    }
    if (x <= cur.val) {
        Node* nex = succ(cur.lch, x);
        if (nex is null || cur.val < nex.val) {
            return cur;
```

```
        }
        return nex;
    }
    return succ(cur.rch, x);
}
```

4.5.3　insert - 要素の挿入

二分探索木に新しく要素を追加することもできます。要素を追加する際は、既存の二分探索木を破壊しないように葉として追加します。

方法はシンプルで、find と同様に二分探索木を降りていけばよいです。



▲図 4.9: insert(5) の動作の様子

図 4.9 はデータ 5 を追加する際のアルゴリズムの動作です。既存の二分探索木の形状が変化していないため、適切な場所を見つけさえすれば頂点を葉として接続するだけで良いことがわかります。

計算量は木の高さを h として $O(h)$ 時間です。

以下に insert のプログラム例を示します。再帰関数を利用して頂点の接続状況を正しく管理しています。insert(cur, x) は「挿入前 cur の部分木だった部分において x を挿入した後の新しい部分木の根」を返却しています。

```
Node* insert (Node* cur, int x) {
    if (cur is null) {
        return new Node(null, null, x);
    }
```

```
    if (x < cur.val) {
        cur.lch = insert(cur.lch, x);
    }
    if (cur.val < x) {
        cur.rch = insert(cur.rch, x);
    }
    return cur;
}
```

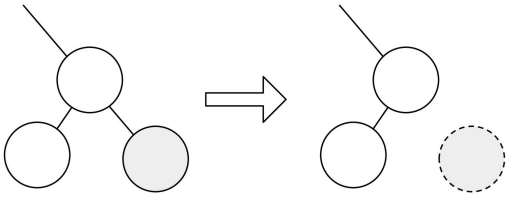
4.5.4　remove - 要素の削除

二分探索木からあるデータを削除することもできます。ただし、内点を削除する場合があるため、アルゴリズムは insert よりも複雑になります。remove も基本的には既存の二分探索木に影響を及ぼさないように操作します。

4.5.4.1　方法 1

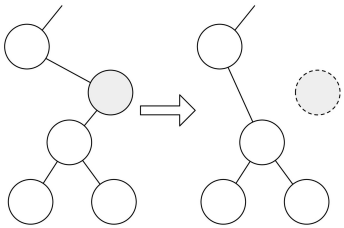
削除対象のデータが木のどこにあるかによって難しさが変わるため、場合分けして考えます。

削除データが葉である場合
単にその葉を切り離せばよいです。図 4.10 は操作を図示したものです。



▲図 4.10: 葉の remove

削除データが右子か左子の片方を持たない場合
その頂点を切り離して、親と残った子を繋げばよいです。図 4.11 は右子を持たない場合の操作を図示したものです。元々二分探索木条件が成り立っていたと仮定するとこの操作が条件を保つことがわかります。

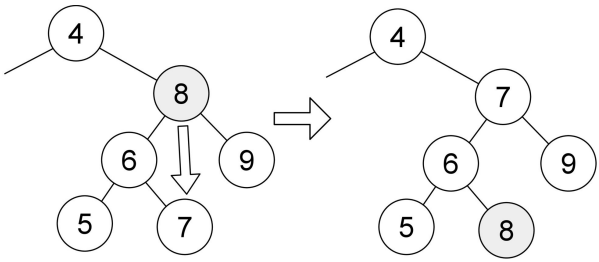


▲図 4.11: 片方の子を持たないデータの remove

それ以外の場合

二分探索木条件から、その頂点と「そのデータ未満の最大のデータを持つ頂点」を入れ替えても木の形と二分探索木条件を保つことができます。まずはそれらの場所を入れ替えて、元々「そのデータ未満の最大のデータを持つ頂点」があった場所で頂点を取り除きます。

仮定から、その頂点は右子を持ちません。[*5]したがって、上記の 2 ケースいずれかに帰着できます。図 4.12 は頂点の入れ替えで削除したいデータがあった位置の二分探索木条件が保たれることを説明する図です。操作後の 8 は葉になっていることがわかります。



▲図 4.12: 両方の子を持つデータ 8 の remove

計算量を考えます。削除データが葉である場合と削除データが右子か左子の片方を持たない場合は、目的の頂点にたどり着くまで木を降りていけばよいだけなので、木の高さを h として $O(h)$ 時間かかります。

それ以外の場合は、削除対象の頂点を発見した後に「そのデータ未満の最大のデータを持つ頂点」を発見するという流れになります。「そのデータ未満の最大のデータを

[*5] もし右子を持っていた場合、「そのデータ未満の最大のデータを持つ頂点」であることに矛盾するからです。

持つ頂点」は削除対象の頂点の左部分木にあるため、全体を通して木を降りていくだけです。よってやはり $O(h)$ 時間になります。

以上より、どの場合でも $O(h)$ 時間になります。

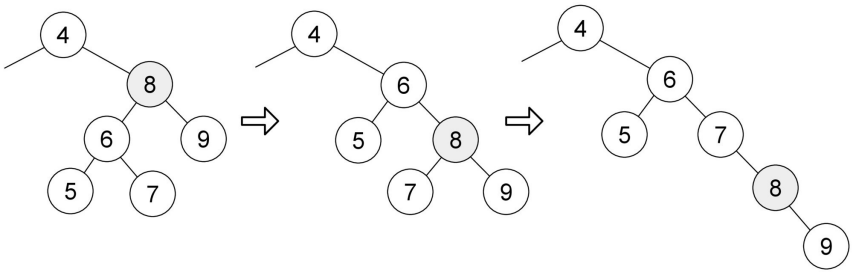
以下に remove のプログラム例を示します。接続状態を正しく保つため再帰関数を使用しているため動作を追うのが少々難しいですが、remove(cur, x) は cur の部分木から x を外し終わった後の新しい部分木の根を返却します。また remove は削除データ未満最大のデータを持つ頂点を取ってくるための関数 removeMost を利用しています。この関数は右子がなくなるまで右子に進み続け、最終到達点の頂点を取り外して返却します。

```
Node* remove1 (Node* cur, int x) {
    if (cur is null) {
        return null;
    }
    if (x < cur.val) {
        cur.lch = remove1(cur.lch, x);
        return cur;
    }
    if (cur.val < x) {
        cur.rch = remove1(cur.rch, x);
        return cur;
    }

    // ケース1、ケース2
    if (cur.lch is null) {
        return cur.rch;
    }
    if (cur.rch is null) {
        return cur.lch;
    }

    // ケース3
    Node*[2] ret = removeMost(cur.lch);

    // curとret[1]を入れ替え（curは外す）
    ret[1].lch = ret[0];
    ret[1].rch = cur.rch;
    return ret[1];
}
```

```
// [新しい根, 左部分木最大のノード（取り外し済み）]
Node*[2] removeMost (Node* cur) {
    if (cur.rch is null) {
        return [cur.lch, cur];
    }
    Node*[2] ret = removeMost(cur.rch);
    cur.rch = ret[0];
    return [cur, ret[1]];
}
```

4.5.4.2　方法 2

方法 1 における「それ以外の場合」で回転を利用ことでより簡単に削除可能です。方法はシンプルで、左子がある限り右回転をし続ければよいです。こうすることで削除対象頂点の左部分木がどんどん小さくなっていき、最終的に空木になります。これで簡単なケースに帰着することができます。図図 4.13 は回転の様子を図示したものです。



▲図 4.13: 削除対象を右回転する様子。左部分木の高さが減っていく様子が確認できる。

右回転 1 回につき左部分木の高さが少なくとも 1 減るため、$O(h)$ 回の回転で簡単なケースに帰着できます。よってこの方法を用いても $O(h)$ 時間です。

以下にプログラム例を示します。左右の子を持つ場合はまず回転し、そのあと削除対象の頂点が移動した方の子に対して再帰関数を呼び出すことで追いかけています。

```
Node* remove2 (Node* cur, int x) {
    if (cur is null) {
```

```
        return null;
    }
    if (x < cur.val) {
        cur.lch = remove2(cur.lch, x);
        return cur;
    }
    if (cur.val < x) {
        cur.rch = remove2(cur.rch, x);
        return cur;
    }


    // ケース1、ケース2
    if (cur.lch is null) {
        return cur.rch;
    }
    if (cur.rch is null) {
        return cur.lch;
    }

    // 右回転 -> 再帰呼び出し
    cur = rotR(cur);
    cur.rch = remove2(cur.rch, x);
    return cur;
}
```

4.5.5　merge - 二分探索木の併合

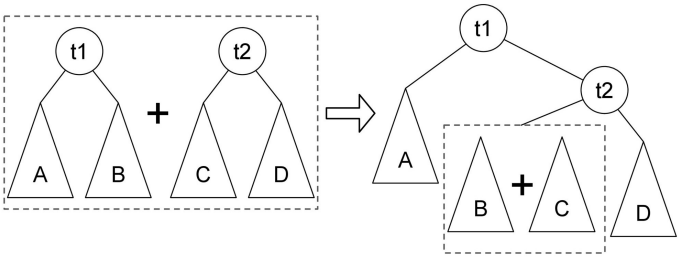
2 つの二分探索木 t_1, t_2 を 1 つに合体することができます。ただし、

$$t_1 \text{ が含むデータ} \leq t_2 \text{ が含むデータ}$$

を満たす必要があります。なお、このような制約のない併合は meld などと呼ばれることがあります。

merge は再帰的に行います。まず t_1 と t_2 の根から片方を任意に選び、もう一方の根を繋ぎます。次に、二分探索木条件から明らかに大小関係が定まる部分木を繋ぎます。最後に残った中央の部分木同士を再帰的にマージします。図 4.14 は t_1 を根とし

て選択した場合の動作を図示したものです。



▲図 4.14: t_1 の根を新しい根とした場合の merge の動作

計算量を考えます。t_1, t_2 の高さを h_1, h_2 とするとき、merge は $O(1)$ 時間の操作を行った後にそれぞれ高さ $h_1 - 1, h_2 - 1$ 以下の木の merge を再帰的に呼び出します。この再帰は少なくとも一方が空木になるまで続くため、計算量は $O(\min(h_1, h_2))$ 時間です。

以下に merge のプログラム例を示します。

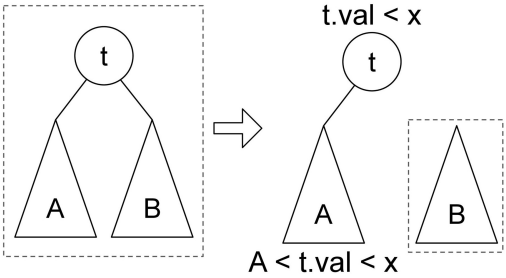
```
Node* merge (Node* t1, Node* t2) {
    if (t1 is null) {
        return t2;
    }
    if (t2 is null) {
        return t1;
    }

    t2.lch = merge(t1.rch, t2.lch);
    t1.rch = t2;
    return t1;
}
```

4.5.6　split - 二分探索木の分割

merge の逆、split も行うことができます。この操作は引数として与えたデータ値 x と二分探索木 t に対して、x 未満のデータをすべて含む二分探索木 t_1 と x 以上のデータをすべて含む二分探索木 t_2 に分解します。

split も再帰的に考えます。まず根と x を比較します。もし根のデータが x 未満で

あった場合、t の根と左子に含まれるデータは x 未満が確定します。そこで、右子を再帰的に split し、その結果を右子に繋ぎます。根のデータが x 以上であった場合は左右反転して同じことを行えばよいです。図 4.15 は根のデータが x 未満であるときの動作を図示したものです。



▲図 4.15: 根のデータが x 未満の場合の split の動作。次は部分木 B を再帰的に split する。

計算量を考えます。split は $O(1)$ 回の操作と再帰的な split の呼び出しを含みます。この再帰的な呼び出しでは t の部分木、つまり高さが 1 以上減った木を渡します。これは空木になるまで続くため、再帰呼び出しの回数は木の高さを h として $O(h)$ 回になります。以上より、合計 $O(h)$ 時間になります。

以下に split のプログラム例を示します。

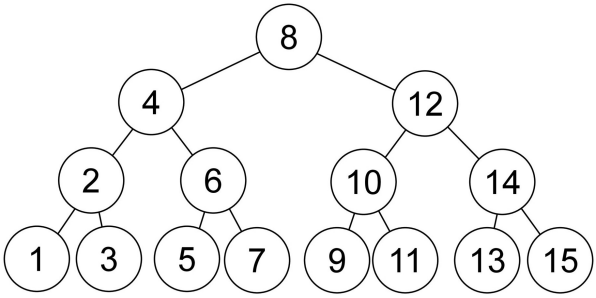
```
Node*[2] split (Node* t, int x) {
    if (t is null) {
        return [null, null];
    }
    if (t.val < x) {
        Node*[2] sp = split(t.rch, x);
        t.rch = sp[0];
        return [t, sp[1]];
    }
    Node*[2] sp = split(t.lch, x);
    t.lch = sp[1];
    return [sp[0], t];
}
```

4.6　性能の解析

二分探索木は各種操作をその木の高さ h に対して $O(h)$ 時間で行えることがわかりました。では木の高さ h はどれくらいになるのでしょうか？

4.6.1　最良の場合

最も高さが低くなるのは**完全二分木**となるときです。完全二分木とは全ての葉が同じ高さで、全ての内点が子を 2 つ持つ二分木のことです。図 4.16 は完全二分木の例です。



▲図 4.16: 完全二分木の形をした二分探索木

この木は根からの距離 x の層に 2^x 個の頂点を持つため、高さが h であるときは

$$2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$$

個の頂点を含みます。[*6]逆に、頂点 n 個を含む場合、その高さは $n \leq 2^{h+1} - 1$ を満たす最小の h ということになります。両辺 1 を足して対数をとると、

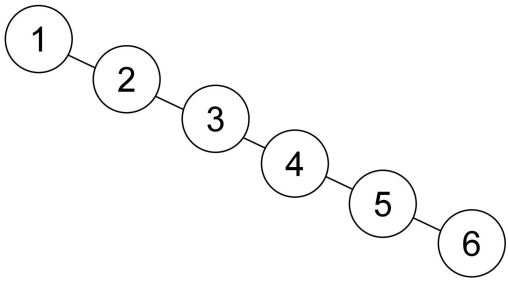
$$\log_2(n+1) \leq h + 1$$

であるため、およそ $\log_2 n$ 程度になります。n の値が完全二分木にできないようなものの場合、葉をいくつかなくしたものが最良の場合になります。

[*6] 等比数列の総和は閉じた式が知られており、公比 $r \neq 1$、項数 n としたとき、$(r^n - 1)/(r - 1)$ となります。

4.6.2 最悪の場合

高さが最も高くなるのは図 4.17 のように直線状になる場合です。この時は頂点数 n に対して高さが $n-1$ となります。



▲図 4.17: 直鎖を成す二分探索木

4.6.3 平均的な場合

こういった極端な形以外にも、二分探索木は様々な形状になります。では平均的な高さはどのくらいになるのでしょうか？ 実は次の事実が知られています。

定理
n 個の頂点をランダムな順番で `insert` し二分探索木を作る場合、任意の頂点の深さの期待値は $O(\log n)$ である。

この定理は、二分探索木の取りうるすべての形状の集合から見ると最悪の場合に近いケースが珍しいということを言っています。つまり、データの挿入順がある程度ランダムであるとわかっているならば、二分探索木に対する操作を非常に高速に行うことができるということです。証明を追ってみましょう。

証明
二分探索木では、データの具体的な値ではなくそれら大小関係のみによって形状が決まります。よって簡単のため、元のデータの代わりに $\{1, 2, \ldots, n\}$ を考えます。まずは次の補題を示します。

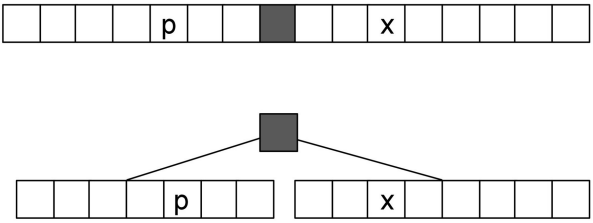
補題
n 個のデータをランダムな順番で `insert` し作った二分探索木において、根からデータ x を持つ頂点までの経路にデータ p を持つ頂点がある確率は $1/(|x-p|+1)$ で

ある。

証明（補題）
簡単のため、$p < x$ を仮定します。$x < p$ でも議論は変わりません。

x までの経路に p が含まれる必要十分条件は、$p, p+1, p+2, \ldots, x$ の中で p が最も早く insert されることです。これは図 4.18 のように、$p+1, p+2, \ldots, x$ が先に選ばれてしまった場合、p と x が別々の部分木に属してしまい、どうやっても経路中に入れなくなってしまうからです。p, x の外側の頂点が選ばれてもこのような分割は発生しないため、$p, p+1, p+2, \ldots, x$ 以外の順番は関係ありません。

逆に、p が最初に insert された場合、二分探索木条件を考えると x にたどり着くには p を通るしかないことがわかります。（証明終わり）



▲図 4.18: p 以外が先に insert された場合の二分探索木の構造。p と x がそれぞれ別の部分木に入ってしまうことがわかる。

深さの解析に戻りましょう。ランダムな順番で insert を行った時のデータ x の深さは確率変数になります。これを D_x としましょう。

また、確率変数 $I_{i,j}$ を、根からデータ i までの経路にデータ j が含まれるなら 1、含まれないなら 0 となる確率変数として定義します。

深さの定義より、D_x は

$$D_x = \sum_{i \neq x} I_{x,i}$$

と表すことができます。この期待値は、期待値の線形性より

$$
\begin{aligned}
E[D_x] &= E\left[\sum_{i \neq x} I_{x,i}\right] \\
&= \sum_{i \neq x} E[I_{x,i}]
\end{aligned}
$$

とできます。$E[I_{x,i}]$ は $I_{x,i}$ の定義を考えると、「ランダムに選んだ insert 順において、x までの経路にデータ i が含まれる確率」と等しくなります。補題より、これは $1/(|x-i|+1)$ です。したがって、

$$
\begin{aligned}
E[D_x] &= \sum_{i \neq x} \frac{1}{|x-i|+1} \\
&\leq 2\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) \\
&\leq 2\left(\int_1^n \frac{1}{x}\mathrm{dx}\right) \\
&\leq 2\ln n
\end{aligned}
$$

となり、深さが $2\ln n$ で抑えられることがわかりました。（証明終わり）

4.6.4 応用: bstSort

ランダムな順序で挿入すると、木の高さが低く抑えられることを示しました。これを利用して、二分探索木を利用したソートを作ることができます。

アルゴリズムはかなり簡単です。まず入力されたデータをシャッフルし、insert して二分探索木を作ります。この二分探索木に対して行きがけ順に探索していくことで小さい要素から順に列挙することができます。

重要なのは入力されたデータを insert する前にシャッフルすることです。これにより、x 個の要素を挿入した時点での木の高さが期待 $\log x$ になるため、n 回の insert の実行時間を合計期待 $O(n \log n)$ 時間に抑えることができます。

行きがけ順による木の走査は $\Theta(n)$ 時間であるため、全体で期待 $O(n \log n)$ 時間です。以下にプログラム例を示します。

```
void bstSort (int[] A) {
    Node* root = null;
    // Aをシャッフル -> 挿入
    randomShuffle(A);
    foreach (v; A) {
        root = insert(root, v);
    }

    // 行きがけ順で走査
    void f (Node* cur, ref int i) {
        if (cur is null) {
            return;
        }
        f(cur.lch, i);
        A[i] = cur.val;
        i++;
        f(cur.rch, i);
    }
    int index = 0;
    f(root, index);
}
```

4.7　参考文献

1. アルゴリズムイントロダクション 第 4 版 第 1 巻、T. コルメン・C. ライザーソン・R. リベスト・C. シュタイン著、浅野 哲夫・岩野 和生・梅尾 博司・小山 透・山下 雅史・和田 幸一訳、近代科学社
   - グラフの定義の参考にしました。
2. アルゴリズムとデータ構造、岩畑 清著、岩波書店
   - グラフの定義、各種操作および図の作成の参考にしました。
3. みんなのデータ構造、Pat Morin 著、堀江慧、陣内佑、田中康隆訳、ラムダノート
   - ランダム二分探索木の性能解析を参考にしました。
4. プログラミングコンテストでのデータ構造 2 〜平衡二分探索木編〜、秋葉拓哉著、https://www.slideshare.net/slideshow/2-12188757/12188757、2025 年 9 月 29 日閲覧
   - split および merge の参考にしました。

あとがき

本書を作成するきっかけは、8 月あたりに Twitter にて UEComic なるイベントが復活するという投稿を見かけたことでした。私は理解したことのアウトプットをするのが好きで、普段ブログ InTheDayDream[*7]を更新しています。このめったにないチャンスに自分の書きたいことを本という形にまとめたい思い、本書を執筆することを決めました。しかし、本作成に対する知識等は全くなかったため、なかなか苦労しました。

本当に何もわからないため、作成に使うツールから選定する必要がありました。いろいろと検討した結果、@kauplan さんが作成した「Re:VIEW Starter」[*8]というツールで作ることにしました。普段の執筆環境である markdown + hugo とは記法や設定などがかなり異なるため、調べながらの作業になり、思ったよりも時間を費やしました。

LaTeX の埋め込み構文で `split` 環境を使うと組版が変な感じになるので、かわりに `aligned` を使うようにしたこと、pdf をそのまま埋め込もうとすると `dvipdfmx` の影響？ で謎のエラーが出るため jpg に変換をかけたりしたことなど、執筆作業は試行錯誤の連続でした。

それでも原稿がだんだん完成していく様子はとても達成感がありました。それに、やはり形になったものを見ると嬉しくなります。

本書は一度 UEComic! Restart!で頒布したものに加筆修正を加えたものになります。UEComic! Restart!から UEComic! 9 までにあまり作業時間が確保できなかったため、掲載を断念したトピックもあります。ですが、掲載できたトピックに関しては満足のいく完成度になったのではないかと思います。

本書をここまで読んでいただきありがとうございました。本書の情報が何らかの役に立ったり、本書をきっかけにして情報発信や同人誌作成をはじめる方が現れてくれれば、それは何よりも嬉しいです。

[*7] https://inthebloom.github.io/

[*8] https://qiita.com/kauplan/items/d01e6e39a05be0b908a1

好きなアルゴリズムとデータ構造について

2025 年 11 月 13 日　発行

著　者　InTheBloom
発行者　InTheBloom
連絡先　nato.rider.smm2@gmail.com
　　https://x.com/uu9782wsedandhp
印刷所　株式会社日光企画

(powered by Re:VIEW Starter)

InTheDayDream