アルゴリズム

担当：
以前は田浦准教授と相田が隔年で担当→近年は毎年相田が担当

目的：
コンピュータアルゴリズムの基礎
≒良いプログラムを書くための先人の知恵を学ぶ

講義資料（未完）：
http://www.aida.t.u-tokyo.ac.jp/~aida/algorithm.pdf

参考書：
- 石畑 : アルゴリズムとデータ構造、岩波講座ソフトウェア科学 3、岩波書店、ISBN4-00-010343-1、1989
- 渋谷 : 東京大学工学教程情報工学アルゴリズム、ISBN978-4-621-30113-5、2016
- 田浦准教授の講義資料：
  http://www.logos.t.u-tokyo.ac.jp/~tau/lecture/computer_software/

評価：
中間レポート＋期末試験の予定

1. 序論
   1.1 アルゴリズムの記法
   1.2 アルゴリズムの正しさの証明
   1.3 アルゴリズムの良さを表す尺度

2. 基本データ構造
   2.1 配列
   2.2 リスト
   2.3 2 重連結リスト
   2.4 配列とリストの比較
   2.5 スタックとキュー：常に O(1)でアクセス可能な使用法

3. 検索
   3.1 線形検索
   3.2 二分検索
   3.3 二分検索木
   3.4 平衡木（Balanced Tree）
   3.5 B 木
   3.6 ハッシュ法

4. 整列
   4.1 O(n^2)の整列アルゴリズム
   4.2 O($n \log n$)の整列アルゴリズム
   4.3 大小比較に基づかない整列アルゴリズム
   4.4 ハードウェアソートアルゴリズム

5. 文字列照合
   5.1 単純なアルゴリズム
   5.2 Knuth Morris Pratt のアルゴリズム
   5.3 正規表現とオートマトン
   5.4 Boyer-Moore のアルゴリズム
   5.5 系列間のマッチング

6. 安定マッチング
   6.1 受入側提案の受入保留アルゴリズム
   6.2 マッチングの安定性
   6.3 学生側提案の受入保留アルゴリズム
   6.4 マッチングの最適性

7. グラフのアルゴリズム
   7.1 グラフの探索
   7.2 最短経路

1. 序論

良いプログラムとは

・ 誤りのないこと早く結果が得られること（デバッグが容易なこと　実行が速いこと）メモリを食わないこと等々

・ 1 回だけ使うプログラムと何度も使うプログラムで自ずと異なる
Σ(プログラムの開発・デバッグに要する時間+Σ プログラムの実行時間)
→1 回だけ使うプログラムのつもりで作ったものを何度も使って問題が生じることが多い（入力制限　メモリ容量　スクリプト）

アルゴリズムとは

・ 問題を解くための手順を（きちんと）定めたもの（石畑）
・ 問題を解くための効率的手順を定式化した形で表現したもの（Wikipedia）

ざっというと：プログラムをもう少し抽象化したもの

・ 2 つの数の大きい方を選ぶ：あたりまえ
・ たくさんの数の中から一番大きいものを選ぶ：あらかじめどう並べておくか
・ たくさんの候補者の中から最も大統領にふさわしい人を選ぶ：いろいろなやり方があり得る

いろいろなアルゴリズム

・ 普通は正しい結果が得られることと停止性を保証
cf. 近似アルゴリズム/確率的アルゴリズム/computational method

・ 多くのアルゴリズムは単一 CPU での逐次処理を想定
cf. 並列アルゴリズム/ハードウェアアルゴリズム

アルゴリズムを学ぶ意義

・ 他人の書いたプログラムを理解し再利用する
・ 効率の良いプログラムを短時間で仕上げる
→先人の知恵を活用する

1.1 アルゴリズムの記法

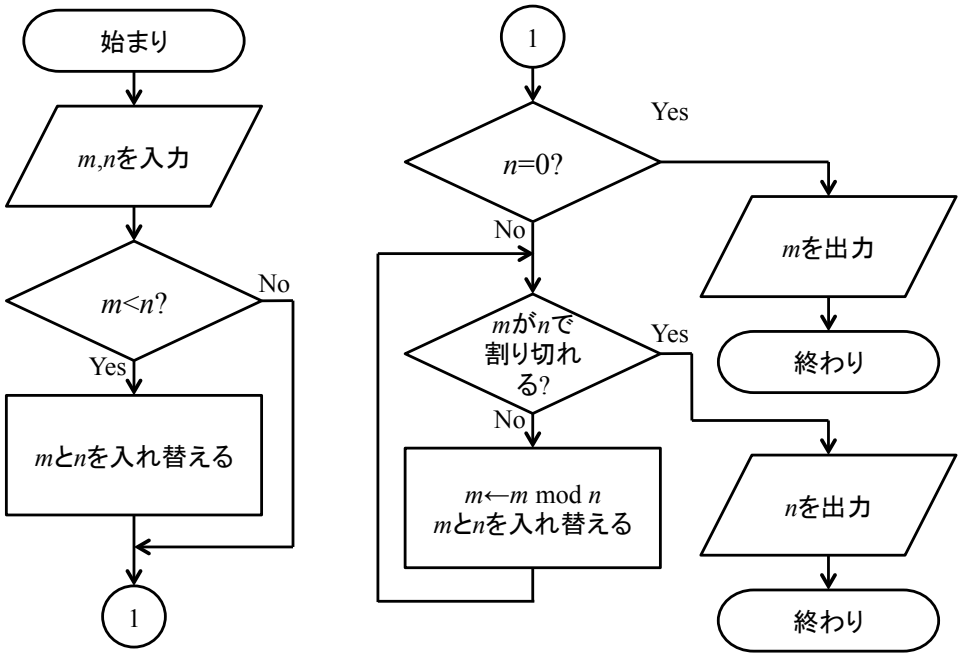
・ 自然言語
・ フローチャート等
・ 疑似コード

・ プログラム

自然言語

例：ユークリッドの互除法（Wikipedia）
1. 入力を m, n（$m \geqq n$）とする。
2. $n = 0$ なら、m を出力してアルゴリズムを終了する。
3. n が m を割り切るなら、n を出力してアルゴリズムを終了する。
4. m を n で割った余りを新たに m とし、更に m と n を取り替えて 3.に戻る。

フローチャート



疑似コード

```
gcd(m,n)
{
    if (m<n)
        SWAP(m,n);
    if (n==0) return m;
    else for (;;) {
        r = m % n;
        if (r == 0)
            return n;
        m=r;
        SWAP{m,n};
    }
}
```

1.2 アルゴリズムの正しさの証明

通常は、

- （停止するとすれば）正しい結果が得られること
  と
- 停止すること
  を分けて証明

正しい結果が得られること

- $m = d \times m' \quad n = d \times n' \quad m'$と$n'$は互いに素
- $r = m - q \times n = d \times \{m' - q \times n'\}$よって$r$も$d$を約数に持つ
- n'と$m' - q \times n'$が公約数d'を持てば
  $n' = d' \times n'' \quad m' - q \times n' = d' \times m''$より
  $m' = d' \times m'' + q \times n' = d' \times m'' + q \times d' \times n'' = d' \times \{m'' + q \times n''\}$
  $\therefore m'$も同じ約数を持ち、m'とn'は互いに素に反する
  $\therefore m$とnの最大公約数=nとrの最大公約数

- 一般に、ループを回る間、ある条件(ループ不変条件)が常に成立していることを証明することが多い
  - 限界的な場合に注意

停止性

- $r < n$
  →rは単調減少
  →ループを回ればいつかは必ず$r == 0$となる

1.3 アルゴリズムの良さを表す尺度

- プログラム開発時のデバッグの容易さ

- 実行時に
  どの程度時間がかかるか→時間計算量
  どの程度メモリを必要とするか→領域計算量
  どの程度入出力を行うか
  どの程度通信を行うか（並列アルゴリズム・分散アルゴリズム）

- しばしばトレードオフの関係

時間計算量

・ 絶対的な数値にはあまり意味がない
　CPU のクロック・コンパイラ等で異なる
・ ループを回る回数、関数呼び出しの回数など
　「プログラムの実行時間の 90%は、プログラム全体の 10%の部分の実行に費やされる」

```
int
gcd1(int m, int n)
{
    int r;

    if (n == 0)
        return m;
    else for (;;) {
        if ((r = m % n) == 0)
            return n;
        m = n;
        n = r;
    }
}

int
gcd2(int m, int n)
{
    int r;

    for ( ; n != 0; r = m % n, m = n, n = r)
        ;
    return m;
}

int
god3(int m, int n)
{
    int r;

    if (n == 0)
        return m;
    else for (;;) {
        while (m >= n)
            m -= n;
        if (m == 0)
            return n;
        r = m, m = n, n = r;
    }
}
```

アルゴリズムの解析

・ 計算量を「問題の大きさ」*n* で表す
　➢　実際の数値、データの個数、等々

- 最悪値
- 平均値

漸近的評価

・ $n \to \infty$の振る舞い

・ オーダ記法

- ある定数 N, c があって$\forall n > N$ について $f(n) < cg(n)$のとき
  $f(n) = O(g(n))$
- ある定数 N, c'があって$\forall n > N$ について $f(n) > c'g(n)$のとき
  $f(n) = \Omega(g(n))$
- $f(n) = O(g(n))$かつ $f(n) = \Omega(g(n))$のとき
  $f(n) = \Theta(g(n)$

n = 1,000 のときの実効時間を 10s とすると…

n	1,000	2,000	5,000	10,000	20,000	50,000	100,000	1,000,000
O(1)	10	10	10	10	10	10	10	10
O(log n)	10	11.0	12.3	13.3	14.3	15.7	16.7	20
O(n)	10	20	50	1'40"	3'20"	8'20"	16'40"	2:46'40"
O(n log n)	10	22.01	61.65	2'13"	4'47"	13'3"	27'47"	5:33'20"
O(n^2)	10	40	4'10"	16'40"	1h6'40"	6h56'40"	1d3:47'	115d17:47'
O(n^3)	10	1'20"	20'50"	2:47'	22:13'	14d11:13'	115d17:47'	316y324d

n	10	15	20	25	30	40	50	60
O(2^n)	10	5'20"	2:51'	3d19:1'	121d8:43'	340y93d	348,421y	3.56×10^8y

領域計算量

・ 以前ほど重要ではなくなったが、ワーキングセットがある大きさを超えると
性能が急激に低下
・ 大きさそのものよりもアクセスの局所性・順次性等が問題

・ 隠れた領域計算量
- 再帰呼び出しの引数・ローカル変数
- n を表現するには log n ビット必要
- （普通は 1 語におさまる範囲で考える）

2. 基本データ構造

2.1 配列

```
struct student_data {
    int  id;
    char name[MAX_NAME_SIZE];
    int  score;
};

struct student_data student_array[MAX_STUDENT_COUNT];
int student_count; /* <= MAX_STUDENT_COUNT */
```

i 番目のデータの参照：O(1)

```
if (i < 0 || i >= student_count)
    ERROR("i: out of range");
else
    student_array[i];
```

データの挿入：O(*n*)

```
insert_student(int i, struct student_data x)
{
    int j;

    if (student_count >= MAX_STUDENT_COUNT)
        ERROR("array full");
    else {
        for (j = student_count, j > i, j--)
            student_array[j] = student_array[j-1];
        student_array[i] = x;
        student_count++;
    }
}
```

データの削除：O(*n*)

```
delete_student(int i)
{
    if (i < 0 || i >= student_count)
        ERROR("i: out of range");
    else {
        for( ; i < student_count-1; i++)
            array[i] = array[i+1];
        student_count--;
    }
}
```

- 各データに「削除済み」フラグを付けるとデータの削除を O(1)で済ますことが可能だが、代わりに *i* 番目のデータの参照が O(*i*)となる

2.2 リスト

```
struct student_list {
    struct student_data data;
    struct student_list *next;
};

struct student_list *student_head;
```

i 番目のデータの参照：O(*i*)

```
struct student_list *
i_th_student(int i)
{
    struct student_list *p;

    if (i < 0 || (p = student_head) == NULL)
        return NULL;
    while (i-- > 0)
        if ((p = p->next) == NULL)
            return NULL;
    return p;
}
```

指定したデータの後ろに新たなデータを挿入：O(1)

```
insert_student(struct student_list *prev, struct student_data x)
{
    struct student_list *p;

    if ((p = (struct student_list *)
              malloc(sizeof(struct student_list))) == NULL)
        ERROR("cannot allocate memory");
    else {
        p->data = x;
        if (prev == NULL) {
            p->next = student_head;
            student_head = p;
        } else {
            p->next = prev->next;
            prev->next = p;
        }
    }
}
```

指定したデータの後ろのデータを削除：O(1)

```
delete_next_student(struct student_list *prev)
{
    struct student_list *p;

    if (prev == NULL) {
        p = student_head;
```

```
        student_head = p-> next;
    } else {
        p = prev->next;
        prev->next = p->next;
    }
    free((void *)p);
}
```

指定したデータを削除：O(n)

```
delete_student(struct student_list *p)
{
    struct student_list *q;

    if (p == NULL)
        ERROR(“delete null”);
    else if ((q = student_head) == p)
        delete_next_student(NULL);
    else do {
        if (q->next == p) {
            delete_next_student(q);
            return;
        }
    } while (q = q->next);
    ERROR(“not in list”);
}
```

2.3 2 重連結リスト

```
struct student_dlist {
    struct student_data data;
    struct student_dlist *next, *prev;
};
```

指定したデータの後ろに新たなデータを挿入：O(1)

```
insert_student(struct student_dlist *prev, struct student_data x)
{
    struct student_dlist *p;

    if ((p = (struct student_dlist *)
              malloc(sizeof(struct student_dlist))) == NULL)
        ERROR("cannot allocate memory");
    else {
        p->data = x;
        if (prev == NULL) {
            p->next = student_head;
            student_head = p;
        } else {
            p->next = prev->next;
            prev->next = p;
        }
        p->prev = prev;
    }
```

```
}
```

指定したデータを削除：O(1)

```
delete_student(struct student_dlist *p)
{
    if (p->prev == NULL)
        student_head = p->next;
    else
        p->prev->next = p->next;
    if (p->next)
        p->next->prev = p->prev;
    free((void *)p);
}
```

プログラミングテクニック：先頭・末尾の特別扱いを避けるには

・ 先頭にダミーのセルを置いておく

```
struct student_list student_head;
```

・ さらに、リストの最後を表すのに NULL ではなく先頭に戻すようにする

```
struct student_list student_head = &student_head;
```

・ さらに、next ポインタがリストの先頭に来るようにすることで、data 部分のメモリ確保を省略

```
struct student_list {
    struct student_list *next;
    struct student_data data;
};
struct student_list *student_head;
```

・ 2 重連結リストの場合→2 重リング

ここで、単に

```
struct student_dlist *student_head, *student_tail;
```

と書いたのでは隣接して取られるとは限らない

```
struct student_dlink {
    struct student_dlist *next, *prev;
};

struct student_dlist {
    struct student_dlink links;
    struct student_data data;
};
```

```
struct student_dlink student_headtail;
#define student_head student_headtail.next
#define student_tail student_headtai1.prev
```

2.4 配列とリストの比較

配列：
全ての要素に同じ時間でアクセス可能
データだけがメモリ上に整然と格納される
あらかじめデータ数に余裕を持ってメモリ領域を確保しておく必要あり
```
struct student_data *student_array;
student_array = malloc(sizeof(struct student_data)*n);
```
データの挿入・削除に手間がかかる

リスト：
メモリ領域は必要の都度確保するのであらかじめデータ数を知る必要なし
データの挿入・削除・並べ替えが比較的容易
データごとにポインタ領域が余分に必要
中間にあるデータにアクセスするにはポインタをたどる必要あり

2.5 スタックとキュー：常に O(1)でアクセス可能な使用法

LIFO (Last In First Out)：スタック(Stack)

・ 配列の場合
常に最後にデータを追加(push)／最後のデータを参照すると同時に削除(pop)

・ リストの場合
常に先頭にデータを追加(push)／先頭のデータを参照すると同時に削除(pop)

FIFO (First In First Out)：キュー(Queue)

・ リストの場合：データの末尾を覚えておいてそこに挿入

```
struct student_list *student_head, *student_tail;

void
enqueue_student(struct student_data x)
{
    struct Student_list *p;

    if ((p = (struct student_list *)
              malloc(sizeof(struct student_list))) == NULL)
        ERROR("cannot allocate memory");
    else {
        p->data = x;
        p->next = NULL;
```

```
        if (student_head == NULL)
            student_head = p;
        else
            student_tail->next = p;
        student_tail = p;
    }
}

struct student_data
dequeue_student(void)
{
    struct student_list *p;
    struct student_data x;

    if (student_head == NULL)
        ERROR("queue empty");
    else {
        p = student_head;
        x = p->data;
        student_head = p->next;
        free((void *)p);
        return x;
    }
}
```

・ 配列の場合：リングバッファ：最後まで行ったら先頭から再利用

```
int first_student, last_student;
#define next_student(x) ((x) < MAX_STUDENT_COUNT-1 ? (x)+1 : 0)

void
enqueue_student(struct student_data x)
{
    if (next_student(last_student) == first_student)
        ERROR("queue full");
    else {
        student_array[last_student] = x;
        last_student = next_student(last_student);
    }
}

struct student_data
dequeue_student(void)
{
    int i;

    if ((i = first_student) == last_student)
        ERROR("queue empty");
    else {
        first_student = next_student(first_student);
        return student_array[i];
    }
}
```

3. 検索

どの要素について検索するか：キー（key）

簡単のため、データ中でキーは重複していないものとする

3.1 線形検索

配列の場合

```
int
linear_search(int score)
{
    int i;

    for (i = 0; i < student_count; i++)
        if (student_array[i].data.score == score)
            return i;
    return -1;
}
```

リストの場合

```
struct student_data *
linear_search(int score)
{
    struct student_list *p;

    for (p = student_head; p; p = p->next)
        if (p->data.score == score)
            return &(p->data);
    return NULL;
}
```

線型検索の計算量

見つからない場合：O(n)

見つかる場合：何番目に見つかるか→どのデータが見つかるか均等とすれば O(n)

データをキーの順に並べておく

データのキーが検索するキーより大きく（あるいは小さく）なった時点でループから抜け出すことが可能

データがキーの順に並んでいるならもっと良い方法がある

番兵

```
struct student_data *
sentinel_search(int score)
{
    int i;

    student_array[student_count].data.score = score;
    for (i = 0; ; i++) {
        if (student_array[i].data.score == score)
            return i < student_count ? &student_array[i]: NULL;
}
```

自己再構成リスト

見つかったデータをリストの先頭に移動

良く検索されるデータがリストの前の方に来る

3.2 二分検索

配列上でデータがキーの順に並んでいるものとする

001	湯川　秀樹	31
002	朝永　振一郎	41
003	川端　康成	59
004	江崎　玲於奈	26
005	佐藤　栄作	53
006	福井　謙一	58
007	利根川　進	97
008	大江　健三郎	93
009	白川　英樹	23
010	野依　良治	84

⇒

009	白川　英樹	23
004	江崎　玲於奈	26
001	湯川　秀樹	31
002	朝永　振一郎	41
005	佐藤　栄作	53
006	福井　謙一	58
003	川端　康成	59
010	野依　良治	84
008	大江　健三郎	93
007	利根川　進	97

```
struct student_data *
binary_search(int score)
{
    int lower_bound, upper_bound, middle;

    lower_bound = 0;
    upper_bound = student_count - 1;
    while (lower_bound <= upper_bound) {
        middle = (lower_bound + upper_bound) / 2;
        if (score == student_array[middle].data.score)
            return &student_array[middle];
        else if (score < student_array[middle].data.score)
            upper_bound = middle — 1;
        else
            lower_bound = middle + 1;
    }
```

```
    return NULL;
}
```

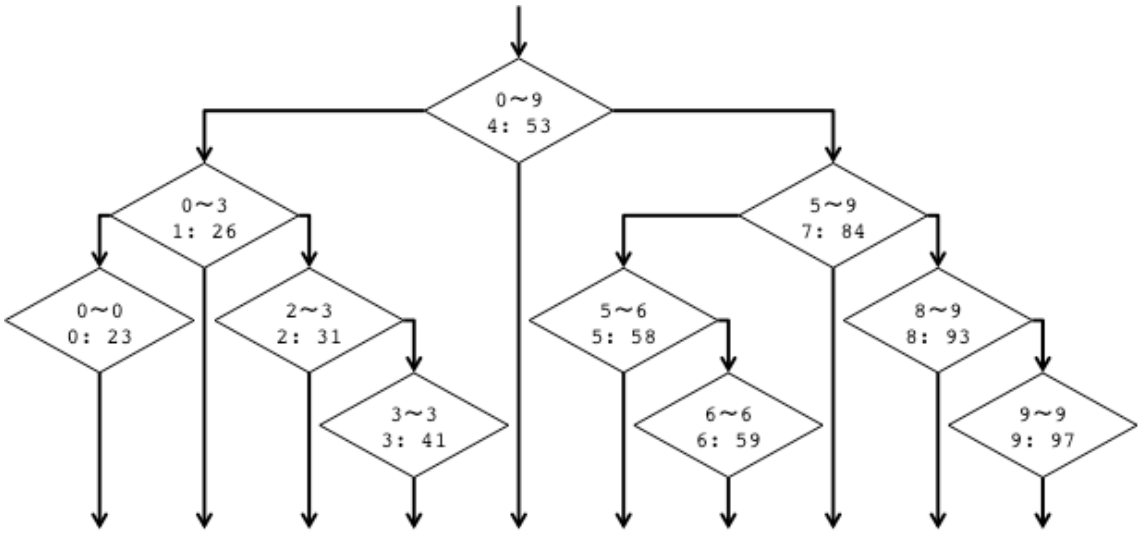
計算量

検索：O(log *n*)

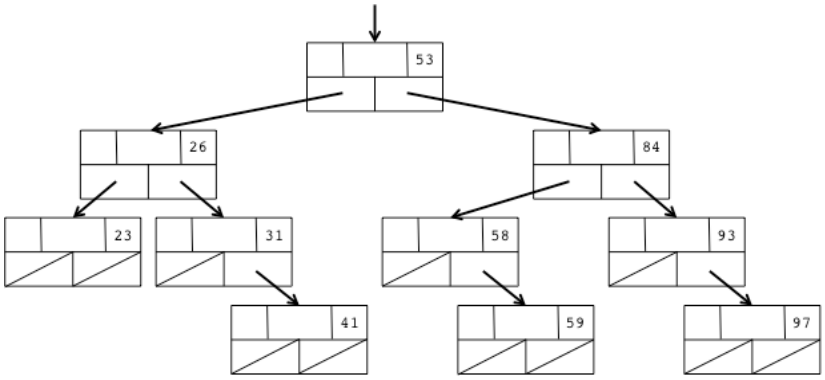
データ追加：挿入位置を検索するのに O(log *n*)、挿入するのに O(*n*)

3.3 二分検索木

データ追加も O(log *n*)でできるようにならないか→リスト的アプローチ




```
struct student_tree {
    struct student_data data;
    struct student_tree *left, *right;
};
```




```
struct student_tree *student_root;

struct student_data *
binary_search(int score)
```

```
{
    struct student_tree *p;

    for (p = student_root; p; )
        if (score == p->data.score)
            return &(p->data);
        else if (score < p->data.score)
            p = p->left;
        else
            p = p->right;
    return NULL;
}
```

どのようにして木を作るか

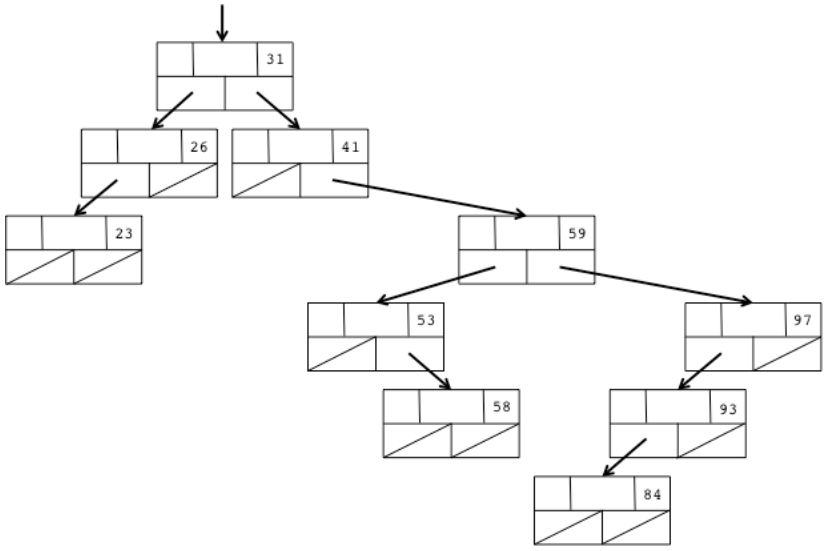
大事なことは全てのノードにおいて

`left` からたどれる全てのノードの `score` はそのノードの `score` より小さい
`right` からたどれる全てのノードの `score` はそのノードの `score` より大きい

が満たされていること。逆にこれが満たされていれば上記の `binary_search` で検索可能

```
void
add_student(struct student_data x)
{
    struct student_tree *p, *q, *r;

    if ((p = (struct student_tree *)
              malloc(sizeof(struct student_tree))) == NULL)
        ERROR("cannot allocate memory");
    else {
        p->data = x;
        p->left = p->right = NULL;
        if ((q = student_root) == NULL)
            student_head = p;
        else for ( ; ; q = r)
            if (score == q->data.score) {
                ERROR("same score");
                return;
            }
            if (score < q->data.score) {
                if ((r = q->left) == NULL) {
                    q->left = p;
                    return;
                }
            } else if ((r = p->right) == NULL) {
                q->right = p;
                return;
            }
    }
}
```



平均計算量

i個のデータからなる木に登録されているデータを検索する際にたどるノードの数：C_i（C_0=0, C_1=1）

最初に登録されたデータが小さい方からi番目のデータだとすると`left`側の木の大きさ：$i-1$, `right`側の木の大きさ：$n-i$。従ってn個のデータ全てについて検索した場合の重み付き平均値は

$$\frac{1}{n}\cdot 1+\frac{i-1}{n}\left(1+C_{i-1}\right)+\frac{n-i}{n}\left(1+C_{n-i}\right)$$
$$=1+\frac{1}{n}\left\{\left(i-1\right)C_{i-1}+\left(n-i\right)C_{n-i}\right\}$$

最初に登録されるデータが n 個のデータのいずれであるかに関する平均を取ると

$$C_n=1+\frac{1}{n^2}\sum_{i=1}^{n}\left\{\left(i-1\right)C_{i-1}+\left(n-i\right)C_{n-i}\right\}$$
$$=1+\frac{2}{n^2}\sum_{i=1}^{n}\left(i-1\right)C_{i-1}$$
$$nC_n=n+\frac{2}{n}\sum_{i=1}^{n}\left(i-1\right)C_{i-1}$$

$nC_n=A_n+1$ とおくと（A_0=−1, A_1=0）

$$A_n + 1 = n + \frac{2}{n}\sum_{i=1}^{n}\left(A_{i-1} + 1\right)$$

$$A_n = n + 1 + \frac{2}{n}\sum_{i=1}^{n} A_{i-1}$$

$$nA_n = n(n+1) + 2\sum_{i=1}^{n} A_{i-1}$$

$$(n-1)A_{n-1} = (n-1)n + 2\sum_{i=1}^{n-1} A_{i-1}$$

$$nA_n - (n-1)A_{n-1} = 2n + 2A_{n-1}$$

$$nA_n - (n+1)A_{n-1} = 2n$$

$$\frac{A_n}{n+1} - \frac{A_{n-1}}{n} = \frac{2}{n+1}$$

$$\frac{A_n}{n+1} - \frac{A_0}{1} = \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{2}$$

$$\frac{A_n}{n+1} = \sum_{i=1}^{n}\frac{2}{i+1} - 1$$

$H_n = \sum_{i=1}^{n}\frac{1}{i}$（調和級数）とおくと

$$\frac{A_n}{n+1} = 2\left(H_n + \frac{1}{n+1} - \frac{1}{1}\right) - 1$$

$$A_n = 2(n+1)H_n - 3n - 1$$

$$C_n = \frac{A_n + 1}{n} = \frac{2(n+1)}{n}H_n - 3$$

$H_n \fallingdotseq \log_e n + \gamma$（$\gamma$ は Euler の定数$\fallingdotseq 0.577$）であることから

$C_n \fallingdotseq 2\log_e n = 1.39\log_2 n$

従って $C_n = O(\log n)$であり、しかも理想的な場合と比べて 39%しか悪くないことがわかる。

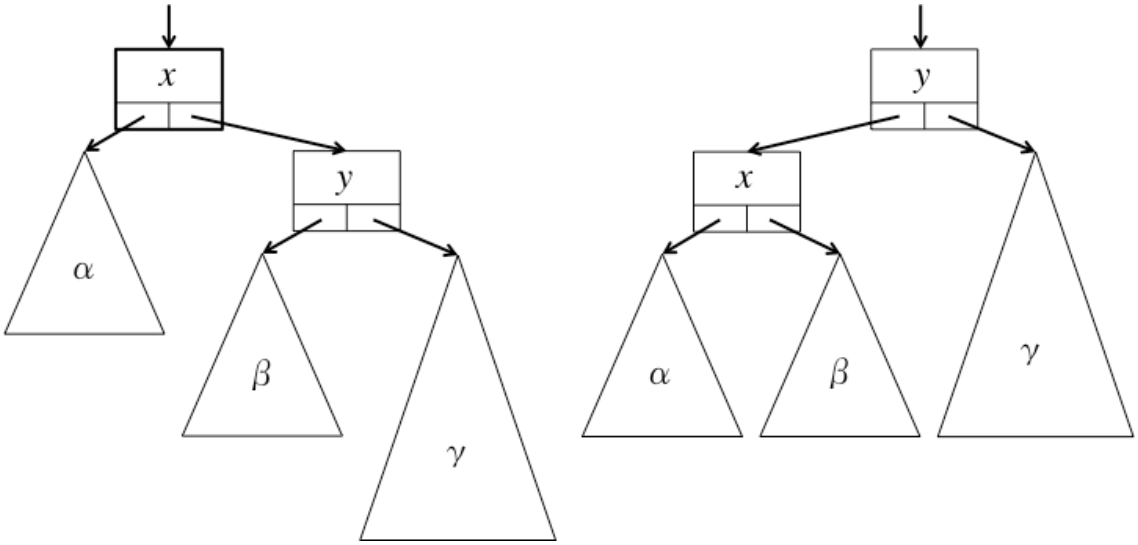
別のキーで検索したいとき

- そのキー用のポインタを設ける
- キー→データ格納位置の表を別に設ける：インデックス

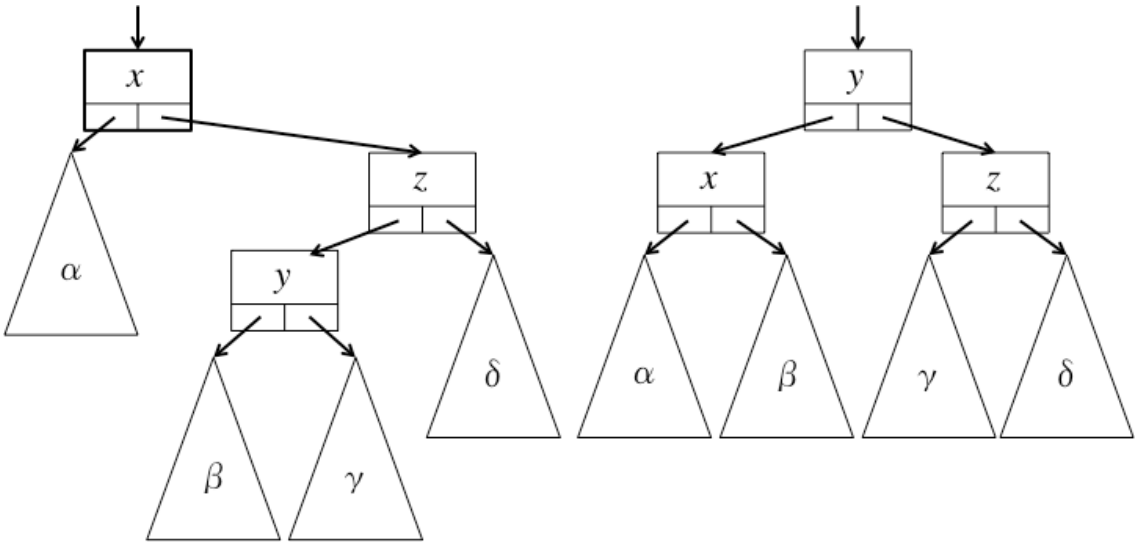
3.4 平衡木（Balanced Tree）

データを追加するたびに O(log *n*)までの手間をかけて木の偏りを直し、高さが O(log *n*)に収まるようにできないか

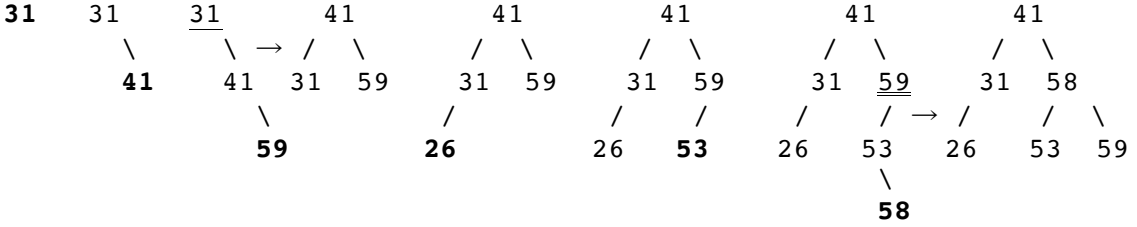
単純回転

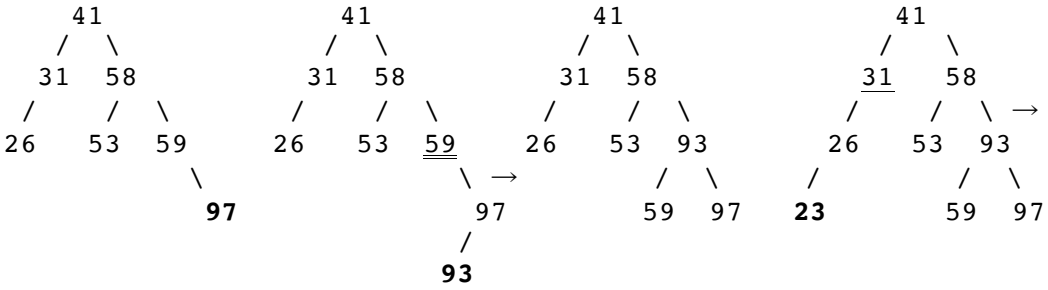


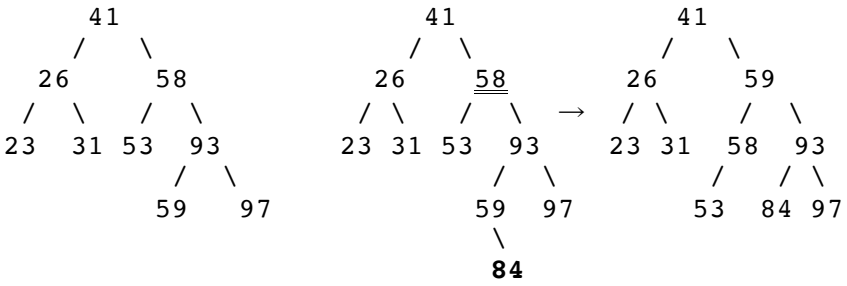
二重回転



平衡木の例：AVL 木：全ての部分木における左右の木の高さを 1 以内に抑える







m_i：高さ i の木を構成し得る最小のノード数（m_0=0, m_1=1）

$m_{i+1} = 1 + m_i + m_{i-1}$
$m_{i+1}+1 = (m_i+1) + (m_{i-1}+1)$

$$m_i + 1 = \frac{5+3\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{2}\right)^i + \frac{5-3\sqrt{5}}{10}\left(\frac{1-\sqrt{5}}{2}\right)^i$$
$$= 1.171 \times 1.618^i - 0.171 \times (-0.618)^i$$

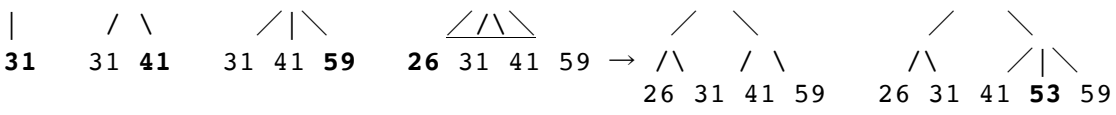
$i \fallingdotseq \log_{1.618} m_i = 1.44 \log_2 m_i$

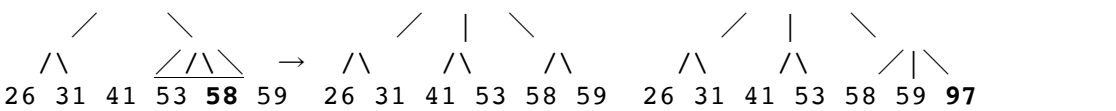
木の高さは最悪でも理想的場合の 1.44 倍に抑えられる

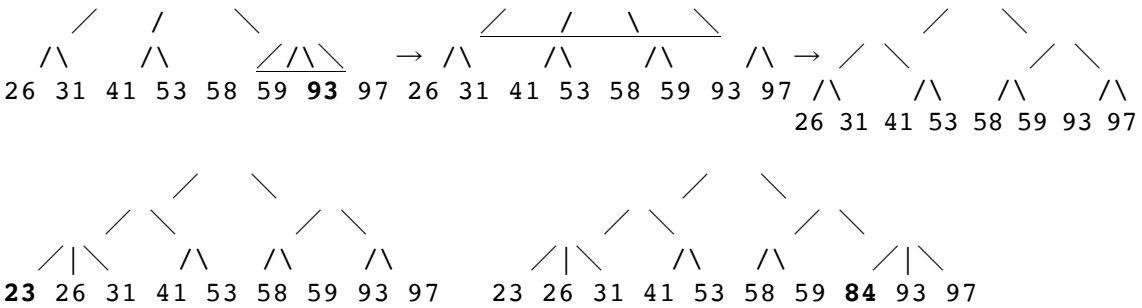
3.5 B 木

各ノードでの最大 m 分岐→根以外での分岐数を ceil(m/2)〜m に抑える

最小：2-3 木







二次記憶のインデックスとして良く用いられる：キーとポインタ合わせて 16 バイトとすれば、ディスク上のブロックサイズを 1024 バイトとして m=64。4 回のディスクアクセスで 64^4=16,777,216 ブロック=17,179,869,184 バイトのディスクをインデックスできる。

3.6 ハッシュ法

これまでの検索法：1 回の比較で m 分岐→$\log_m n$ 回の比較が必要

$m > n$ にできないか？　cf. トライ（trie）

キーからそれがどこに格納されているかを O(1)で与える関数：ハッシュ関数
例：文字コードを足し合わせる

問題点：異なるキーの値に対してハッシュ関数の値が一致してしまう（衝突）
cf. 暗号理論における一方向性関数

チェイン法

ハッシュ関数の値ごとにリストを作る：$n \to n/m$ の効果

開番地法（Open Addressing）

・ 線形走査法

衝突が起きたら次に入れる（$h_{i+1}(x) = \{h_i(x) + 1\} \bmod m$）：クラスタが生じやすい

・ 均一ハッシュ法

ハッシュ関数を m 通り用意して $h_1(x)$で衝突が起きたら $h_2(x)$, $h_3(x)$, ...を順に用いる。

例：
$h_{i+1}(x) = \{h_i(x) + g(x)\} \bmod m$ （二重ハッシュ）
$h_{i+1}(x) = \{h_i(x) + 2i - 1\} \bmod m$（二次ハッシュ）

大きさ m の表にすでに n 個のデータが登録されていて n+1 番目のデータを追加するとき、i 番目のハッシュ関数で空きが見つかる確率を p_i とすると

$$p_1 = \frac{m-n}{m}$$

$$p_2 = \frac{n}{m} \times \frac{m-n}{m-1}$$

$$p_3 = \frac{n}{m} \times \frac{n-1}{m-1} \times \frac{m-n}{m-2}$$

$$p_k = \frac{{}_{m-k}\mathrm{C}_{n-k-1}}{{}_{m}\mathrm{C}_{n}}$$

$$\sum_{k=1}^{n+1} k p_k = \frac{m+1}{m-n+1} \equiv P_k$$

$\alpha = \frac{n}{m}$ とおくと $P_k \fallingdotseq \frac{1}{1-\alpha}$ であり、検索において見つからない場合の平均検索回数もこれに等しい。

見つかる場合の平均検索回数

$$\frac{1}{n}\left(P_0 + P_1 + \cdots + P_{n-1}\right) = \frac{m+1}{n}\left(H_{m+1} - H_{m-n+1}\right) \fallingdotseq \frac{1}{\alpha}\log_e \frac{1}{1-\alpha}$$

→O(1)と称する。

重要な点：データを削除する場合には印を付けておいて、検索の際にそこに当たった場合には衝突が起きたものとして扱わなければならない。

4. 整列

キーの昇順あるいは降順にデータを並べ替える

4.1 O(n^2)の整列アルゴリズム

4.1.1 バブルソート

隣り合うデータで大小順が正しくないものがあったら入れ替える操作を大小順が正しくないデータが無くなるまで繰り返す

```
31 41 59 26 53 58 97 93 23 84

31 41 26 59 53 58 93 97 23 84

31 26 41 53 59 58 93 23 97 84

26 31 41 53 58 59 23 93 84 97

26 31 41 53 58 23 59 84 93 97

26 31 41 53 23 58 59 84 93 97

26 31 41 23 53 58 59 84 93 97

26 31 23 41 53 58 59 84 93 97

26 23 31 41 53 58 59 84 93 97

23 26 31 41 53 58 59 84 93 97
```

```
void
bubblesort()
{
    int i;

    do for (flag = i = 0; i < student_count - 1; i++)
        if (student_array[i].score > student_array[i+1].score) {
            SWAP(student_array[i], student_array[i+1]);
            i++;    /* not necessary */
            flag = 1;
        }
    while (flag);
}
```

アルゴリズムが停止することの証明：それぞれの要素より大きな数が左側にいくつあるかを足しあわせたものはループを回るごとに狭義単調減少

4.1.2 選択法

最も小さな要素を選んでは未整列の要素のうち一番左のものと入れ替える

```
31 41 59 26 53 58 97 93 23 84

23|41 59 26 53 58 97 93 31 84

23 26|59 41 53 58 97 93 31 84

23 26 31|41 53 58 97 93 59 84

23 26 31 41|53 58 97 93 59 84

23 26 31 41 53|58 97 93 59 84

23 26 31 41 53 58|97 93 59 84

23 26 31 41 53 58 59|93 97 84

23 26 31 41 53 58 59 84|97 93

23 26 31 41 53 58 59 84 93|97
```

```
void
selectionsort()
{
    int i, j, k, m;

    for (i = 0; i < student_count - 1; i++) {
        k = i;
        m = student_array[i].score;
        for (j = i+1; j < student_count; j++)
            if (student_array[j].score < m) {
                k = j;
                m = student_array[j].score;
            }
        if (k != i)
            SWAP(student_array[i], student_array[k]);
    }
}
```

比較回数：常に $n(n-1)/2$

4.1.3 挿入法

すでに整列された部分列を右から見ていって挿入すべき場所を見つける

```
 31|41 59 26 53 58 97 93 23 84
   ^
 31 41|59 26 53 58 97 93 23 84
      ^
 31 41 59|26 53 58 97 93 23 84
^
 26 31 41 59|53 58 97 93 23 84
         ^
 26 31 41 53 59|58 97 93 23 84
```

```
             ^
 26 31 41 53 58 59|97 93 23 84
                  ^
 26 31 41 53 58 59 97|93 23 84
                     ^
 26 31 41 53 58 59 93 97|23 84
^
 23 26 31 41 53 58 59 93 97|84
                        ^
 23 26 31 41 53 58 59 84 93 97
```

```
void
insertionsort()
{
    int i, j;
    struct student_data x;

    for (i = 1; i < student_count; i++) {
        x = student_array[i];
        for (j=i-1; j>=0 && student_array[j].score > x.score; j--)
            student_array[j+1] = student_array[j];
        if (j != i-1)
            student_array[j+1] = x;
    }
}
```

比較回数の平均値：$n(n-1)/4$
すでに整列済みである場合には$(n-1)$回の比較で済む

4.2 O(n log n)の整列アルゴリズム

n 個のデータの大きさの並び方の順序：$n!$通り

データを大きさの順序に並べ替えるためには $n!$のどの並び方であるかに応じた並べ替えをしなければならない→$n!$通りのどれであるか区別する必要がある

大小比較では 1 度に 2 通りに分類できる

$a_1 < a_2$	$a_1 > a_2$	
$a_1\ a_2\ a_3$		$a_1 < a_3$
$a_1\ a_3\ a_2$	$a_2\ a_1\ a_3$	
$a_3\ a_1\ a_2$	$a_2\ a_3\ a_1$	$a_1 > a_3$
	$a_3\ a_2\ a_1$	

$n!$通りに分類するのに必要な比較回数：$\log_2 n!$

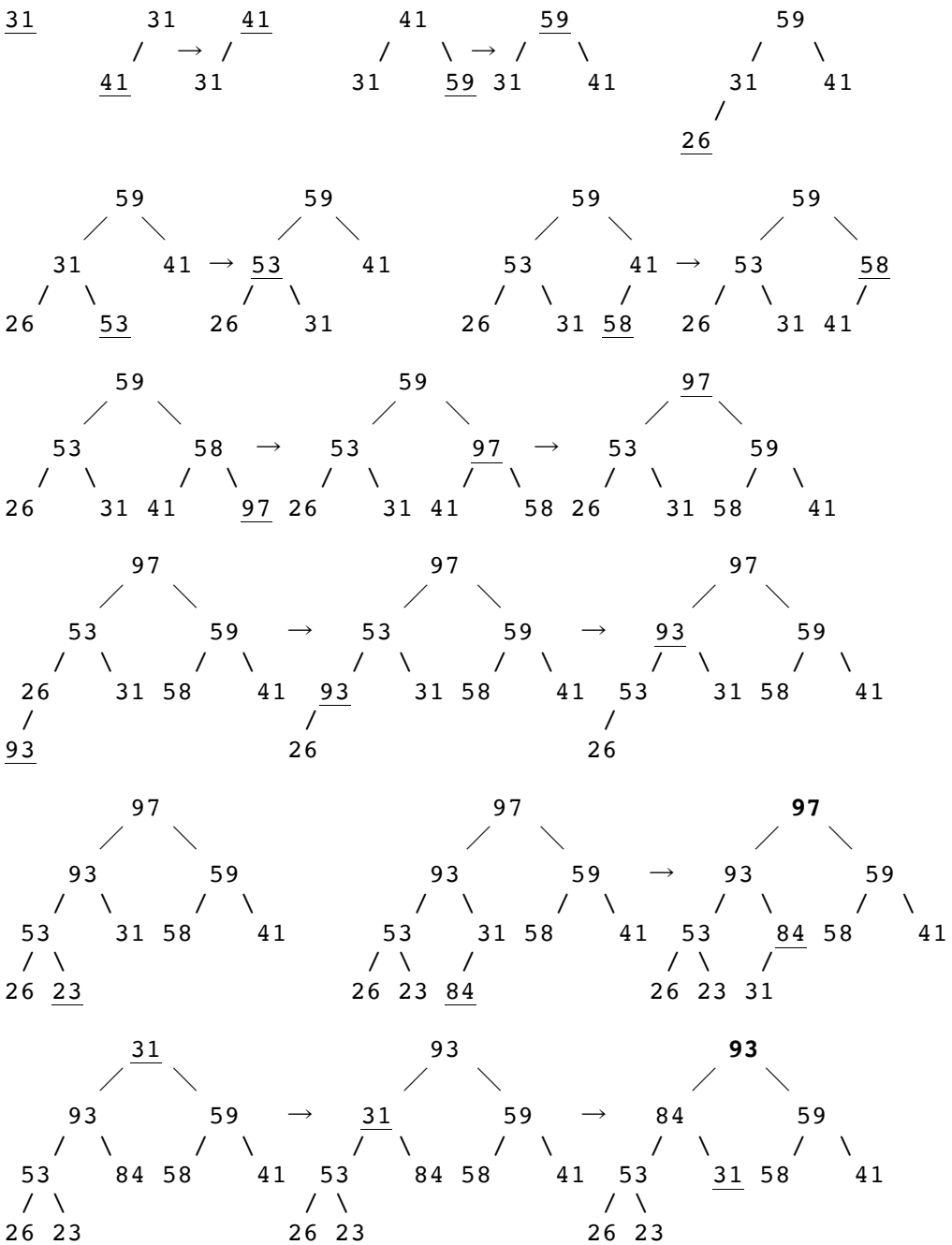
$n! \approx \sqrt{2\pi n}\, n^n e^{-n}$（Stirling の公式）より $\log_2 n! = O(n \log n)$

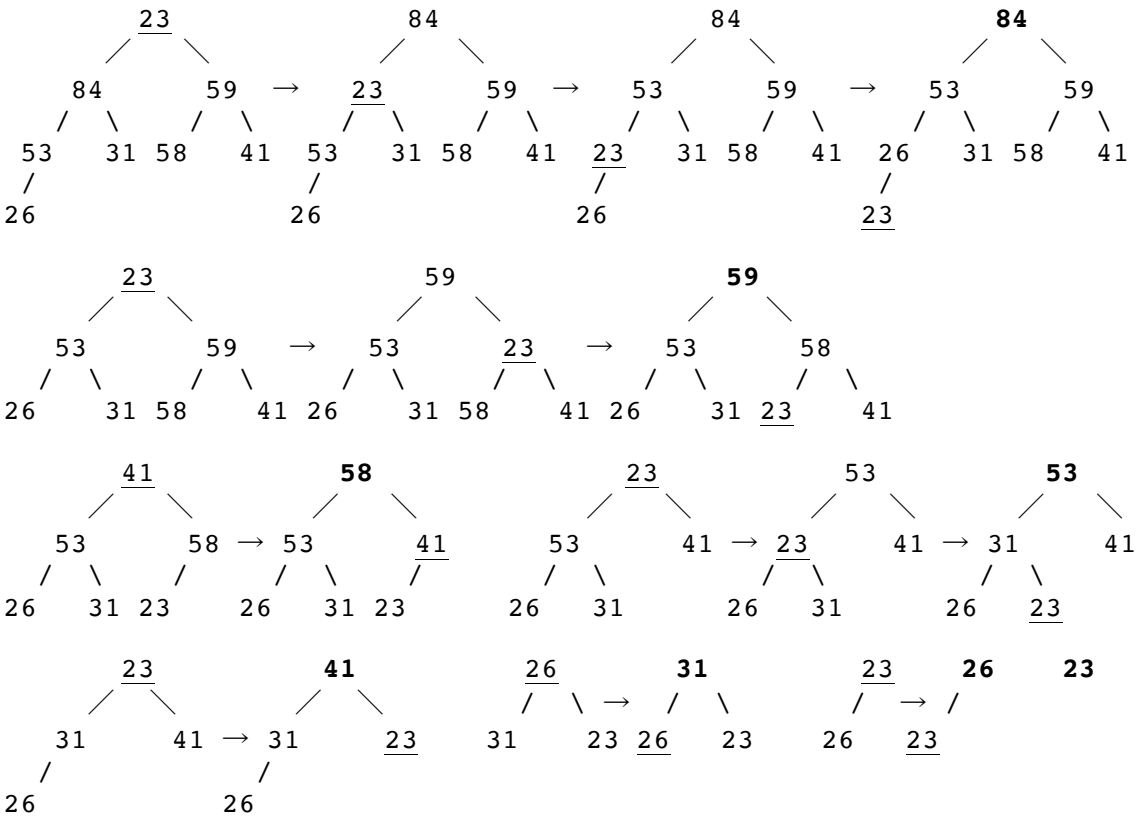
平衡木に n 個のデータを順に登録し、小さいものから順に取り出せば $O(n \log n)$ を実現可能ではあるが、もっと効率の良い方法は？

4.2.1 ヒープソート（Heap Sort）

ヒープ（Heap）：部分順序付き木を配列上に表現したもの

選択法における最大値（最小値）の検索を $O(\log n)$にできる





4.2.2 クイックソート（Quick Sort）

分割（Partition）操作に基づく整列アルゴリズム
ピボット（pivot）以下の要素を左側に、ピボット以上の要素を右側に集める
左から見ていってピボットより大きい要素を見つけ、右から見ていってピボットより小さい要素を見つけ、それらを入れ替える
→左右のそれぞれについて再帰呼び出し

```
31 41 59 26 53 58 97 93 23 84
      >                 <
31 41 23 26 53 58 97 93 59 84
            ><
31 41 23 26|53|58 97 93 59 84
   >     <     >        <
31 26 23 41|53|58 84 93 59 97
       < >           >  <
31 26 23|41|53|58 84 59 93 97
>     <              < >
23 26 31|41|53|58 84 59|93 97
   ><               >  < ><
23|26|31|41|53|58 59 84|93|97
                   < >
23|26|31|41|53|58 59|84|93|97
                 ><
23|26|31|41|53|58|59|84|93|97
```

実用上最高速

・ ピボットの選び方が重要

全ての要素がピボットより小さい or 全ての要素がピボットより大きいとダメ
→最小値≦ピボット≦最大値

中央値がベスト
①先頭、中央、末尾の要素の中央値
②先頭の要素と末尾の要素の平均値
③ランダムな位置の要素の値を使う
④中央の要素の値を使う
⑤先頭の要素の値を使う

最悪：1 つ（あるいは 2 つ）とそれ以外→$O(n^2)$

・ 分割アルゴリズムの詳細

ピボットと一致する要素で止まるか通過するか
→止まることにすれば番兵の原理で必ず止まる

入れ替えの後、その要素から続けるべきか次から続けるべきか
→同じ要素から続けると無限に繰り返す可能性

```
partition(int left, int right)
{
    int pivot, i, j;

    if (left >= right)
        return;
    pivot を選択;
    for (i = left, j = right; ; ) {
        while (student_array[i].score < pivot) i++;
        while (student_array[j].score > pivot) j--;
        if (i > j) break;
        else if (i == j) {
            i++; j--; break;
        } else {
            SWAP(student_array[i], student_array[j]);
            if (++i > --j) break;
        }
    }
    partition(left, j);
    partition(i, right);
}
```

n 個のデータが k 個と$(n-k)$個に分割される確率を $p_n(k)$、$(k-1)$個と 1 個と$(n-k)$個に分割される確率を $q_n(k)$とすれば、平均計算量 Q_n は

$$Q_n = \sum_{k=1}^{n-1} p_n(k)\big((n+2)+Q_k+Q_{n-k}\big)+\sum_{k=1}^{n} q_n(k)\big((n+1)+Q_{k-1}+Q_{n-k}\big)$$

簡単のため $p_n(k)=\frac{1}{n-1}$, $q_n(k)=0$ とすれば

$$Q_n = \big(n+2\big)+\frac{2}{n-1}\sum_{k=1}^{n-1} Q_k$$

$$\big(n-1\big)Q_n = \big(n+2\big)\big(n-1\big)+2\sum_{k=1}^{n-1} Q_k$$

$$\big(n-2\big)Q_{n-1} = \big(n+1\big)\big(n-2\big)+2\sum_{k=1}^{n-2} Q_k$$

$$\big(n-1\big)Q_n-\big(n-2\big)Q_{n-1} = 2n+2Q_{n-1}$$

$$\big(n-1\big)Q_n-nQ_{n-1} = 2n$$

$$\frac{Q_n}{n}-\frac{Q_{n-1}}{n-1} = \frac{2}{n-1}$$

$$\frac{Q_{n-1}}{n-1}-\frac{Q_{n-2}}{n-2} = \frac{2}{n-2}$$

$$\vdots$$

$$\frac{Q_2}{2}-\frac{Q_1}{1} = \frac{2}{1}$$

$$\frac{Q_n}{n}-\frac{Q_1}{1} = 2\left(\frac{1}{n-1}+\frac{1}{n-2}+\cdots+\frac{1}{1}\right) = 2H_{n-1}$$

$$Q_n = 2nH_{n-1} \approx 2n\log_e n = 1.39\log_2 n$$

3 サンプルの中央値を使うとすれば $p_n(k)\approx\frac{3}{2}\left\{1-4\left(\frac{k-n}{2n}\right)^2\right\}$ 、$Q_n \fallingdotseq 1.19\log_2 n$

4.2.3 マージソート（Merge Sort）

外部ソート向き

マージ（Merge）併合
すでに整列された複数の列を先頭から順に見ていって 1 つの列にまとめる
→O(*m*+*n*)

・ バイナリマージソート

```
31|59|53|97|23
41|26|58|93|84
```

```
31 41|53 58|23 84
26 59|93 97|

26 31 41 59|23 84
53 58 93 97

26 31 41 53 58 59 93 97
23 84

23 26 31 41 53 58 59 84 93 97
```

・ 自然マージソート

```
31 41 59|93
26 53 58 97|23 84

26 31 41 53 58 59 97
23 84 93

23 26 31 41 53 58 59 84 83 97
```

4.3 大小比較に基づかない整列アルゴリズム

4.3.1 バケットソート

キーの値に応じて m 個に分類→それぞれを整列→つなぎ合わせ

$n \log n$ は下に凸なので、分類・つなぎ合わせの手間が $O(n \log n)$より少なければ全体を一括して整列するより速くできる

4.3.2 基底ソート（Radix Sort）

まず 1 の位について分類

```
0
1 31 41
2
3 53 93 23
4 84
5
6 26
7 97
8 58
9 59
```

つなげる→`31 41 53 93 23 84 26 97 58 59`

次いで 10 の位について分類

```
0
1
2 23 26
3 31
4 41
5 53 58 59
6
7
8 84
9 93 97
```

つなげる→`23 26 31 41 53 58 59 84 93 97`

安定性：同じキーの値をもった要素は元の順序を保つ

O(*mn*)　*m*: キーの長さ

4.4 ハードウェアソートアルゴリズム

O(*n*)程度の比較・入れ替えを同時に行うことができる前提のアルゴリズム

ソーティング要素（Sorting Element）：比較と入れ替えを組み合わせたもの

縦方向に何段接続する必要があるか　cf. バブルソート：*n*−1

4.4.1 バイトニックマージソート

バイトニック（Bitonic）双調　cf. Monotonic 単調
n を周期とする関数が最小値・最大値以外に極値を持たないこと

f(*x*)が *n* を周期とする双調関数であるとき
$g(x) \equiv \min(f(x), f(x+n/2))$, $h(x) \equiv \max(f(x), f(x+n/2))$はいずれも *n*/2 を周期とする双調関数であり、かつ $\max g(x) \leqq \min h(x)$
→分割操作として用いることができる

入力をどのようにして双調にするか：*n*/2 要素のソータ 2 つを逆順に接続

2^k 入力のバイトニックマージネットワークの段数：*k*

2^k 入力のバイトニックソートネットワークの段数を b_k とすると
$b_1=1$ かつ $b_k=b_{k-1}+1+k \rightarrow b_k=k(k+1)/2$

4.4.2 Odd-Even マージソート（Odd-Even Merge Sort）

前半 2^{k-1} 個と後半 2^{k-1} 個がいずれも整列されているとき、奇数個目からなる部分列と偶数個目からなる部分列をそれぞれ整列すると

```
31 41 59 26 53 58 97 93|23 84 62 64 33 83 27 95

26 31 41 53 58 59 93 97|23 27 33 62 64 83 84 95

26 41 58 93|23 33 64 84
31 53 59 97|27 62 83 95

23 26 33 41 58 64 84 93
27 31 53 59 62 83 95 97
```

前半上側の要素（例えば 41）に着目すると、その要素よりも大きな要素の数は

前半上側 i、前半下側 i+1、後半上側 j、後半下側 j　　または
前半上側 i、前半下側 i+1、後半上側 j、後半下側 j+1　のいずれか。

上側、下側をそれぞれ整列した後では

前者の場合：上側 i+j、下側 i+j+1
後者の場合：上側 i+j、下側 i+j+2

で、後者の場合に左下の要素のみと逆転が生じる。

段数はバイトニックソートと同じだが、バイトニックソートでの振り分けに 2^{k-1} 個のソーティング要素が必要であるのに対して、Odd-Even マージソートで奇数偶数それぞれの整列 後の突き合わせは $2^{k-1}-1$ 個のソーティング要素で済むので、ソーティング要素の総数では Odd-Even マージソートの方が少ない。

5. 文字列照合

5.1 単純なアルゴリズム

1 文字ずつずらしながら比較

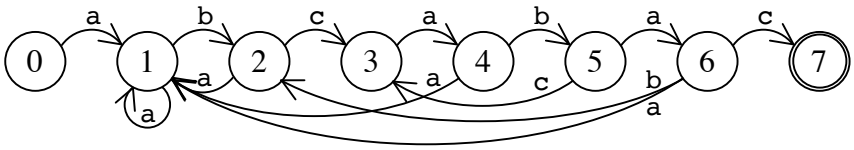
```
a b a b c a b a b c a b a c
| | x
a b c a b a c
  x
  a b c a b a c
    | | | | | x
    a b c a b a c
      x
      a b c a b a c
        x
        a b c a b a c
          | | x
          a b c a b a c
            x
            a b c a b a c
              | | | | | | |
              a b c a b a c
```

5.2 Knuth Morris Pratt のアルゴリズム

パターンの途中まで一致したときにパターンを何文字ずらして良いか、あるいは直前何文字一致していると見なせるかをあらかじめ表にしておく

```
                  ずらし量  何文字一致
~~a~~
  a               1         -1
a ~~b~~
  a               1         0
a b ~~c~~
    a             2         0
a b c ~~a~~
        a         4         -1
a b c a ~~b~~
        a         4         0
a b c a b ~~a~~
      a b c       3         2
a b c a b a ~~c~~
          a b     5         1
```

```
a b a b c a b a b c a b a c
| | x
a b c a b a c
    | | | | | | x
    a b c a b a c
                | | | | | | |
                a b c a b a c
```



5.3 正規表現とオートマトン

・ 正規表現

a	1 文字	
PQ	連結	パターン *P* の後ろにパターン *Q* が続く
P|*Q*	選択	パターン *P* またはパターン *Q*
*P**	閉包	パターン *P* の 0 回以上の繰り返し

必要に応じて括弧を使用（閉包>連結>選択）

・ オートマトン

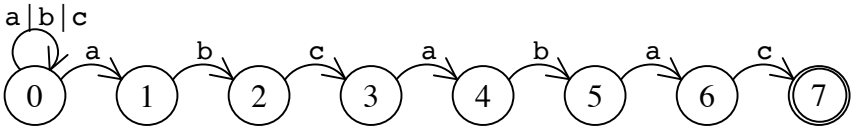
初期状態（initial state）から始めて、文字列から 1 文字読み込むごとに状態を遷移し、全ての文字を読み込んだ時点で最終状態（final state）にいればその文字列を受理（accept）、そうでなければ拒否（reject）

・ 非決定性オートマトン

ある状態において同じ文字に対する遷移が複数あり得る→どちらに遷移しても良い→全ての文字を読み込んだ時点で最終状態にいるような遷移があれば受理

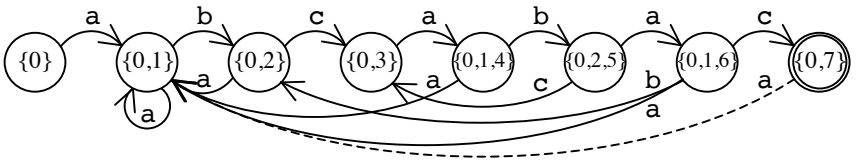
・ ε 遷移

入力を読み込まずに遷移しても良い



状態	a	b	c
0	0, 1	0	0
1		2	
2			3
3	4		
4		5	
5	6		
6			7

状態集合	a	b	c
{0}	{0, 1}	{0}	{0}
{0, 1}	{0, 1}	{0, 2}	{0}
{0, 2}	{0, 1}	{0}	{0, 3}
{0, 3}	{0,1,4}	{0}	{0}
{0,1,4}	{0, 1}	{0,2,5}	{0}
{0,2,5}	{0,1,6}	{0}	{0, 3}
{0,1,6}	{0, 1}	{0,2}	{0, 7}
{0, 7}	{0, 1}	{0}	{0}



5.4 Boyer-Moore のアルゴリズム

文字列の後ろから照合する。
不一致だったときに入力中の文字に応じて何文字ずらせるかと
KMP と同様に途中まで一致したときに何文字ずらせるかの表を作成し、大きい方を取る

```
            文字                ずらし量
              a
  a b c a b a c                 1
              b
    a b c a b a c               2
              c
        a b c a b a c           4
              x（それ以外）
                a b c a b a c   7

                                ずらし量
          ~~a~~ c
        a b c a b a c           4
        ~~b~~ a c
                a b c a b a c   7
      ~~a~~ b a c
                a b c a b a c   7
    ~~c~~ a b a c
                a b c a b a c   7
  ~~b~~ c a b a c
                a b c a b a c   7
~~a~~ b c a b a c
                a b c a b a c   7


a b a b c a b a b c a b a c
              x
a b c a b a c
                x
    a b c a b a c
                  x
        a b c a b a c
                    x
          a b c a b a c
              | | | | | | |
              a b c a b a c
```

5.5 系列間のマッチング

2 つの系列がどの程度似ているか

・ 編集距離
第 1 の系列に対して
1 要素挿入・1 要素削除・1 要素置き換え
を何回行うことで第 2 の系列にできるか

・ 最長共通部分系列

第 1 の系列の部分系列であって第 2 の系列の部分系列でもあるもの（共通部分系列）の中で最も長いもの

・ 動的計画法（Dynamic Programming）によるマッチング

$D_{i,j}$: 第 1 の系列の i 番目までの要素からなる部分系列と第 2 の系列の j 番目までの要素からなる部分系列の距離

$D_{i,j} = \min\{D_{i-1,j-1} + (x_i$ を y_j に変更するコスト),

$D_{i-1,j} + (x_i$ を削除するコスト),

$D_{i,j-1} + (y_j$ を追加するコスト)}

```
a b c a b a c
a c a b c b a
```

		a	c	a	b	c	b	a
	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
b	2	1	2	3	2	3	4	5
c	3	2	1	2	3	2	3	4
a	4	3	2	1	2	3	4	3
b	5	4	3	2	1	2	3	4
a	6	5	4	3	2	3	4	3
c	7	6	5	4	3	2	3	4

6. 安定マッチング

ex. 大学入試、進学選択、卒論配属、就職活動

学生側の希望と受入側の希望の多対 1 のマッチング

YES の決断と NO の決断を同時に行わなければならないことが問題

受入保留アルゴリズム（Deferred-Acceptance Algorithm）
NO の決断は直ちに行うが YES の決断は最後になるまで行わない

6.1 受入側提案の受入保留アルゴリズム

1. 受入側は、志望した学生のうち、受入可能でかつ定員までの順位の学生に対して内定を出す。
2. 複数の受入側から内定をもらった学生は、その中で最も志望順位の高い受入側以外を辞退する。
3. 辞退により欠員の生じた受入側は、次点者から順に受入可能でかつ定員までの志望者に対して追加の内定を出す。
4. 追加の内定が出なくなるまで 2 以下を繰り返す。
5. 辞退されていない内定者を最終的な受入対象者とする。

m : 学生の数、n : 受入側の数
q_x : 受入側 x の定員
$r_a(x)$: 学生 a が受入側 x に付ける志望順位（$1 \leqq r_a(x) \leqq n+1$）
$R_x(a)$: 受入側 x が学生 a に付ける順位（$1 \leqq R_x(a) \leqq m+1$）
$\mu(a)$:（ステップ 2 終了時点での）学生 a の内定先（$1 \leqq \mu(a) \leqq n+1$）
$M = \{(a, \mu(a))\}$
$\rho(x) = \max_{(a,x)\in M} R_x(a)$

アルゴリズムの停止性
アルゴリズム開始時点で$\forall a\ \mu(a)=n+1$ とすれば、ループを回る間、すべての a について $r_a(\mu(a))$は広義単調減少かつ$\Sigma r_a(\mu(a))$は狭義単調減少。よって停止する。

6.2 マッチングの安定性

(i) $\forall (a,x)\in M$ について $r_a(x)\neq n+1$ かつ $R_x(a)\neq m+1$（Individual Rationality）
(ii) $r_a(x) < r_a(\mu(a))$ かつ $R_x(a)\neq m+1$ かつ x の定員に空きがある（$|\{(a,x)\in M\}| < q_x$）ことはない。
(iii) $r_a(x) < r_a(\mu(a))$ かつ $R_x(a) < \rho(x)$ を満たす(a,x)の組（これをブロッキングペアという）は存在しない

(i)は自明。(ii)については、定員に空きがあり、$R_x(a)\neq m+1$ であるなら、途中の

どこかの段階で x は a に内定を出し、どこかの段階で a がそれを辞退しているはずである。 (iii)についても、$R_x(a) < \rho(x)$であるなら、x は順位順に内定を出しているので、途中のどこかの段階で x は a に内定を出し、どこかの段階で a がそれを辞退しているはずである。いずれの場合も、辞退した時点で $r_a(\mu(a)) < r_a(x)$であり、$r_a(\mu(a))$は広義単調減少なので、最終的にも必ず $r_a(\mu(a)) < r_a(x)$となる。

・安定なマッチングは 1 通りとは限らない

6.3 学生側提案の受入保留アルゴリズム

1. 学生は、第一志望の受入側に対して志望を出す。
2. 受入側は、志望者のうち受入不能な者および順位が定員を超す者を拒絶する。
3. 拒絶された学生は、次の志望順位の受入側に対して志望を出す。
4. 新たな志望が出なくなるまで 2 以下を繰り返す。
5. 拒絶されなかった志望者を最終的な受入対象者とする。

6.4 マッチングの最適性

あるマッチングが学生側にとって最適であるとは、各学生の受け入れ先の志望順位 $r_a(\mu(a))$が、他のどんな安定なマッチング M'における志望順位 $r_a(\mu'(a))$と比べても $r_a(\mu(a)) \leqq r_a(\mu'(a))$を満足していること。

・最適なマッチングは 1 通りしかない。

・学生側提案の受入保留アルゴリズムは、学生側にとって最適なマッチングを与える。

学生 a が受入側 x に安定受入可能：$(a, x) \in M$ となる安定なマッチング M が存在すること

学生側提案の受入保留アルゴリズムで拒絶される学生は安定受入可能でない（数学的帰納法）
アルゴリズムのある時点までに拒絶された学生はすべて安定受入不能と仮定。すなわち、全ての学生に関して、その時点での内定先より小さな志望順位の受入先には安定受入不能。
次のステップで学生 a が受入側 x から拒絶されたとすると $R_x(b_1) < \ldots < R_x(b_{q_x}) = \rho(x) < R_x(a)$。もし学生 a が x に受け入れられる安定なマッチングがあったとすると、学生 b_1 から b_{q_x} のうち少なくとも 1 人は現時点での順位より大きな志望順位の受入側に受け入れられることになり、その学生と受入側 x がブロッキングペアとなって、マッチングが安定であることに反する。

・学生側提案の受入保留アルゴリズムが用いられている場合、各学生は、真の志望順位と異なる志望順位を申告することで得をすることはない。（耐戦略性）

7. グラフのアルゴリズム

頂点 vertex：$v_0, \ldots, v_{n-1}$
辺 edge：頂点 v_i と頂点 v_j を繋ぐ辺 e_{ij}　重み w_{ij} がついている場合あり

有向グラフ：辺に始点と終点
無向グラフ：辺に向きがない $e_{ij} \equiv e_{ji}$：多くの場合両向きの辺があるのと同等

多くの場合
　同一頂点を繋ぐ辺は考えない
　同じ始点終点間を繋ぐ辺は 1 本だけ⇒集合として定式化できる

頂点の数を n とすると辺の数 m の最大値 $n(n-1)$

密なグラフ：$m = O(n^2)$
疎なグラフ：$m \leqq O(n)$

グラフの表現法：辺の始点、終点、重み

密なグラフ→2 次元配列
疎なグラフ→始点ごとのリスト、全体を 1 つのリスト、重みに基づくヒープ等

経路：頂点 v_i から頂点 v_j に至る経路 $p_{ij} \equiv e_{ij} \mid p_{ik}e_{kj}$

連結⇔非連結：無向グラフにおいて任意の頂点と他の任意の頂点を繋ぐ経路が存在するか否か

強連結：有向グラフにおいて任意の頂点から他の任意の頂点に至る経路が必ず存在する
弱連結：有向グラフにおいて辺の向きをなくした無向グラフが連結

7.1 グラフの探索

与えられた条件を満たす経路が存在するか

どのように系統立てて調べるか

調べるべき経路の数の上限
迷路（最短経路）：$(n-1)\{1 + (n-2)\{2 + (n-3)\{\ldots\}\}\}$
一筆書き：$\min(m, (n-1)) \times \min((m-1), (n-1)) \times \ldots \times 2 \times 1$

あらかじめ経路を列挙するのではなく、候補となる経路を作成しながら条件を満たすかどうか調べる

7.1.1 深さ優先探索

右手を壁に触れながら迷路を進む：経路を先に進んでみて、行き止まりか以前来た頂点に再度到達するなど、先に進んでも意味がなければ引き返して別の経路を探す。

```
depth_first_search(path)
{
    if (path が求める条件を満たしている)
        return ACCOMPLISHED;
    else if (path を延長できないか延長しても意味がない)
        return TERMINATED;
    for (path の終点 v_k を始点とする辺 e_kj のうちまだ辿っていないものについて)
        if (depth_first_search(path+e_kj) == ACCOMPLISHED)
            return ACCOMPLISHED;
    return EXHAUSTED;
}
```

候補となる経路を LIFO で調べることに相当。

7.1.2 幅優先探索

辺の数の少ない経路から順に探す : 候補となる経路を作成しながら FIFO で調べる。

```
breadth_first_search(path)
{
    enqueue(path, 経路候補キュー);
    while (経路候補キューが空でない) {
        path = dequeue(経路候補キュー);
        if (path が求める条件を満たしている)
            return ACCOMPLISHED;
        else if (path を延長できないか延長しても意味がない)
            continue;
        for (path の終点 v_k を始点とする辺 e_kj のうちまだ辿っていないものについて)
            enqueue(path+e_kj, 経路候補キュー);
    }
    return EXHAUSTED;
}
```

7.2 最短経路

頂点 v_i から頂点 v_j に至る最短経路 : 始点が頂点 v_i、終点が頂点 v_j である経路のうち辺の重みの和が最も小さいもの（1 つとは限らない）

特定の頂点から特定の頂点に至る最短経路だけを効率よく見出すアルゴリズムは知られていない

7.2.1 Dijkstra 法

特定の頂点から他の全ての頂点に至る最短経路を求めるアルゴリズム。

$V = \{v_0, v_1, \dots, v_{n-1}\}$：頂点の集合
w_{ij}：v_iから v_jに至る辺の重み（$\geqq 0$, 辺が存在しないときには∞）
S_0：v_0からの最短経路（および最短距離）が既知の頂点の集合
S_1：v_0からの最短経路（および最短距離）が未知の頂点の集合
d_{ij}：頂点 v_iから頂点 v_jに至る最短距離の上限

初期値
$S_0 = \{v_0\}$
$S_1 = V - S_0 = V - \{v_0\}$
$d_{00} = 0$
$d_{0j} = \infty$（$1 \leqq j < n$）

繰り返し部分

```
while (S1≠φ) {
    for (vi∈S0, vj∈S1である全ての i,j に対して)
        d0j = min(d0j, d0i+wij);
    最小の d0j を与える j（vj∈S1）について{
        S0 = S0∪{vj};
        S1 = S1−{vj};
    }
}
```

正しいことの証明：背理法。

7.2.2 Floyd-Warshall 法

全ての頂点から全ての頂点に至る最短距離を、途中経由する頂点に関する動的計画法により求める。

初期値
$d_{ii} = 0$
$d_{ij} = w_{ij}$（e_{ij}が存在する場合）
$d_{ij} = \infty$（e_{ij}が存在しない場合）

繰り返し部分
第 k ステップでは途中で$\{v_0, \dots, v_k\}$のみを経由する最短距離を求める

```
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            dij = min(dij, dik + dkj);
```

辺の重みの中に負のものがあっても構わない。

7.3 最小木（Minimum Spanning Tree）

無向グラフにおいて、全ての頂点を繋ぐ部分グラフのうち辺の重みの和が最も小さいもの（1 通りとは限らない）。

7.3.1 Prim のアルゴリズム

初期値
$S_0 = \{v_0\}$
$S_1 = V - S_0 = V - \{v_0\}$
$T = \phi$

繰り返し部分
```
while (S1≠φ) {
    最小の wij を与える i,j（vi∈S0, vj∈S1）について{
        T = T ∪{eij};
        S0 = S0 ∪{vj};
        S1 = S1 −{vj};
    }
}
```

正しいことの証明：背理法。

7.3.2 Kruskal のアルゴリズム

辺を重みの昇順にソートし、辺の重みの小さいものから順に採用（ただし辺の両端の頂点を繋ぐ経路が既に存在する場合を除く）。

初期化
$S = \{\{v_0\}, \{v_1\}, \ldots, \{v_{n-1}\}\}$：部分木の集合
辺を重みの昇順にソート

繰り返し部分
```
while (|S| > 1) {
    最小重みの辺の両端の頂点が S の同じ要素に属していればスキップ;
    最小重みの辺を最小木に加える;
    最小重みの辺の両端の頂点が属する S の要素を合併する;
}
```

7.4 最大流（Maximum Flow）

辺の重みが正である有向グラフにおいて、各辺 e_{ij} に流れる流量 f_{ij} の最大値を w_{ij}

とする。

$$\forall i,j f_{ij} \leqq w_{ij}$$

始点 v_s および終点 v_t 以外における外部との出入りはないものとして

$$\forall i(i \neq s,t) \sum_j f_{ij} = \sum_j f_{ji}$$

始点 v_s から終点 v_t に流すことのできる流量

$$\sum_j f_{sj} = \sum_j f_{jt}$$

の最大値を求める。

残余グラフ

フローに対して、各辺の重みが $r_{ij} = w_{ij} - f_{ij} + f_{ji}$ となるよう（必要に応じて）辺を付け加えたグラフ。

残余グラフにおいて、始点 v_s から終点 v_t に至る辺の重みが全て正の経路が存在しないことが、そのフローが最大流であることの必要十分条件。

Ford-Fulkerson 法

流量 0 のフローから始めて、残余グラフに始点 v_s から終点 v_t に至る辺の重みが全て正の経路が存在しなくなるまで、見つかった経路に沿って、経路上の辺の重みの最小値のフローを付け加える。

w_{ij} が無理数である場合などにアルゴリズムが停止しない可能性がある。

（幅優先探索により）辺の数が最小の経路を選ぶようにすれば必ず停止する（Edmonds-Karp 法）。

8. ゲームにおける最善手の選択

二人零和有限確定完全情報ゲーム

二人：三人以上だと自分だけの力では勝敗を決められない
零和：両方とも勝ち、両方とも負けはなし
有限：ゲームが無限に続くことがあると読み切れない
確定：サイコロの目などの偶然に左右されない
完全情報：相手のもっているカードがわからない、ということもない

実は二人が最善を尽くすとしたらどちらが勝つかは決まっている

strongly solved: 全ての局面からの手順が既知
weakly solved: 初手からの手順と（それ以外の局面に対する）戦略が既知
ultraweakly solved: 初手からの最善手順のみ既知

盤面（＋持ち駒）の情報だけで局面は確定（本当は例外あり）

局面の数

三目並べ（○×ゲーム）
3^9=19,683?
${}_9C_1+{}_9C_1\times{}_8C_1+{}_9C_2\times{}_7C_1+{}_9C_2\times{}_7C_2+{}_9C_3\times{}_6C_2+{}_9C_3\times{}_6C_3+{}_9C_4\times{}_5C_3+{}_9C_4\times{}_5C_4+{}_9C_5$
=9+9×8+36×7+36×21+84×15+84×20+126×10+126×5+126
=9+72+252+756+1,260+1,680+1,260+630+126
=6,045

チェッカー
500,995,484,682,338,672,639

オセロ
$2^4\times3^{60}=6.78\times10^{29}\rightarrow10^{60}$

チェス
10^{110}

将棋
先手玉の位置 81×後手玉の位置 80×飛車の位置（先先・後後: $2\times{}_{80}C_2$ (=${}_{79}C_2$+79+1)+先後 79×(78+1)+1×(79+1)）×角の位置×...
10^{226}（10^{224}=阿伽羅）

囲碁
$3^{361}=10^{172}\rightarrow10^{360}$

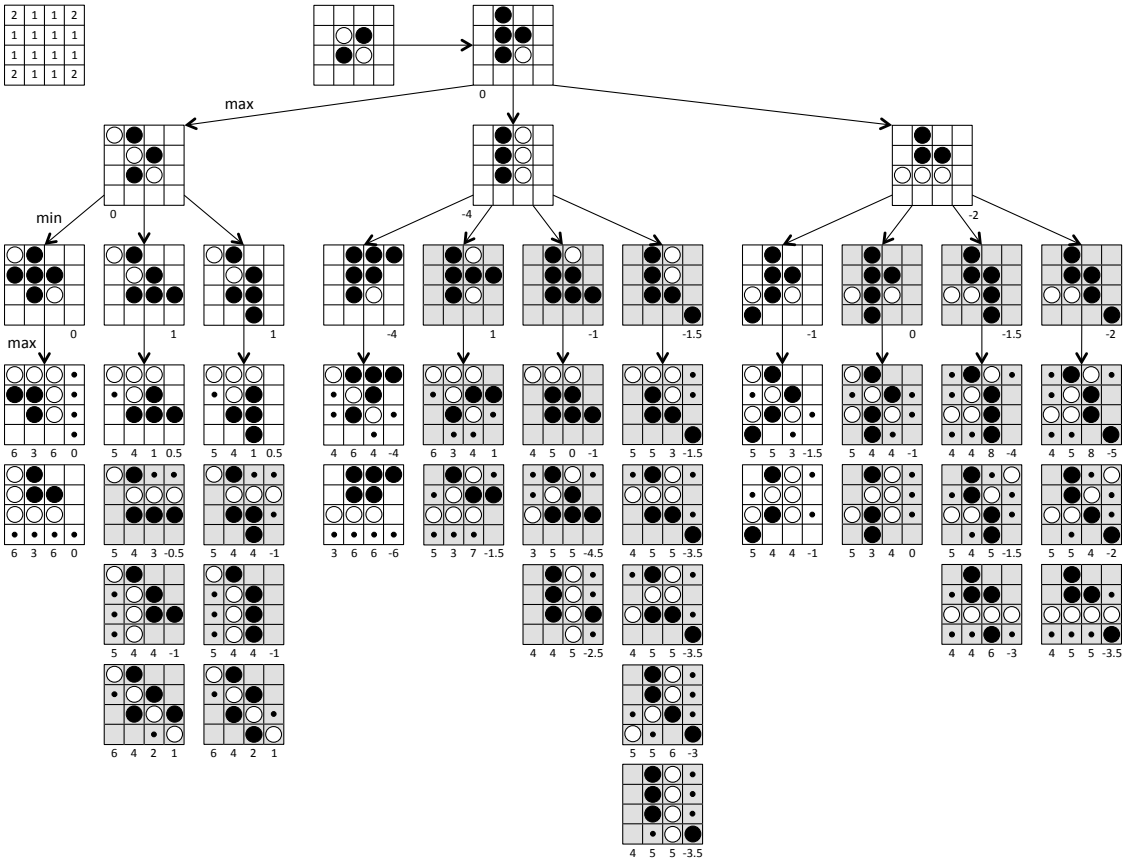
8.1 ゲームの木と mini-max 原理

・ ゲームの木
局面をノードとし、それぞれの局面において指すことが可能な手をエッジとして局面間をつないだグラフ

・ mini-max 原理
相手は常に最善の手を打ってくると考えるべき
cf. 期待値の最大化

・ 静的評価関数
先読みはせずに盤面の良さを定量的に評価
持ち駒の種類、数、位置
持ち駒の効き
次に打つことのできる手の数
等々

従来プログラマが勘と経験により関数形を定めていたが、機械学習を用いることにより劇的に強くなった



8.2 α-β 法に基づく枝刈り

あるノードの評価値が定まることで親ノードの評価値がその評価値より大きい（または小さい）ことが確定→その親ノードは祖父母ノードに採用されないことが確定→弟妹ノードを探索する必要がなくなる

探索を続ける範囲の最小値を α、最大値を β として（当初 $\alpha = -\infty$, $\beta = +\infty$）

```
int
max_node(struct node *node, int depth, int alpha, int beta)
{
    if (depth == 0)
        return static_value_of(node);
    forall (child = child_of(node)) {
        alpha = max(alpha, min_node(child, depth-1, alpha, beta));
        if (alpha > beta)
            return alpha;
    }
    return alpha;
}

int
min_node(struct node *node, int depth, int alpha, int beta)
{
    if (depth == 0)
        return static_value_of(node);
    forall (child = child_of(node)) {
        beta = min(beta, man_node(child, depth-1, alpha, beta));
        if (alpha > beta)
            return beta;
    }
    return beta;
}
```

・ 正解に近い部分から探索すると枝刈りの効率が良い
→静的評価関数をまず適用して有望な枝から探索

深さ一定→水平線効果→局面の安定性に基づきどこまで読むか変える

反復的深化法

9. 難しい問題

9.1 問題の難しさ

・ アルゴリズムの存在し得ない問題

例：プログラムの停止性判定

```
for (x = 1, y = 1, z = 1, n = 3 から x+y+z+n の小さいもの順に) {
    for (xn = 1, yn = 1, zn = 1; i < n; i++) {
        xn *= x; yn *= y; zn *= z;
    }
    if (xn + yn == zn) {
        printf("%d %d %d %d\n", x, y, z, n);
        exit(0);
    }
}
```

プログラムの動作をシミュレート→プログラムが停止するならその手続きも終了するが、プログラムが停止しない場合、その手続きも停止しない。

・ 問題の大きさの多項式時間で解ける問題（P 問題 Polynomial）：やさしい問題

・ 問題の大きさの多項式時間では解けない問題：難しい問題

例：EXPSPACE 問題：領域計算量が問題の大きさの指数関数
拡張正規表現（連結 PQ 選択 $P|Q$ 閉包 P^*に平方 P^2 を加えたもの）の同一性判定

・ 難しいかどうかわからない問題

9.2 NP 問題（Non-deterministic Polynomial）

各ステップにおいていくつかの可能性の中からどれか適当なものを選んで計算を繰り返す

どれを選べば良いのかわかれば（神託 oracle）多項式時間で計算できる
（答えが与えられればそれが正しいことは多項式時間で確認できる）

どれを選べば良いかわからず、全ての組み合わせを試すと問題の大きさの指数関数の時間がかかる

・ 充足可能性判定
n 変数の論理式を真とするような論理変数の真偽値の組み合わせが存在するか

・ Hamilton 閉路問題

n個の頂点から成るグラフにおいて全ての頂点をちょうど1回ずつ通る閉路が存在するか

・ 巡回セールスパーソン問題（判定問題）
n個の頂点から成る重み付きグラフにおいて、全ての頂点を経由する部分グラフで辺の重みの和が与えられた値以下になるものが存在するか

・ k-クリーク問題
無向グラフGと整数kが与えられたときに、k個の頂点から成る完全グラフがGの部分グラフとして含まれるか

・ 部分和問題
自然数の集合Sと自然数tが与えられたときに、Sの部分集合でその総和がちょうどtになるものが存在するか

・ ナップザック問題（判定問題）
n個の品物の集合があり、それぞれの品物の重さはw_i, 価値はv_iである。重さの和がW以下であり、価値の和がV以上となるような部分集合は存在するか

・ 2次不定方程式
$ax^2+by=c$が正整数解を持つか

9.3 多項式時間還元可能とNP完全

ある問題Aの記述を問題の大きさの多項式時間で問題Bの記述に変換でき、問題Bを解いた際の解を多項式時間で問題Aの解に変換できるなら、問題Aは問題Bに多項式時間還元可能（Polynomial time reducible）という

問題Bが多項式時間で解ければ問題Aも多項式時間で解ける

問題Aの難しさは問題B以下

簡単な例：判定問題→最適化問題

・ 巡回セールスパーソン問題（最適化問題）
n個の頂点から成る重み付きグラフにおいて、全ての頂点を経由する部分グラフで辺の重みの和が最小となるものを見出せ
→与えられた重みの和が最小値以上ならばyes、最小値未満ならばno

・ ナップザック問題（最適化問題）
n個の品物の集合があり、それぞれの品物の重さはw_i, 価値はv_iである。重さの和がW以下である部分集合の中で、価値の和を最大とするものを見出せ

・ ハミルトン閉路問題→巡回セールスパーソン問題
全ての辺の重みを 1 かつ辺の重みの和が頂点の数以下となるものを見出す

NP 困難（NP-hard）
全ての NP の問題以上難しい問題

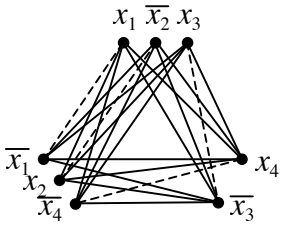
NP 完全（NP-complete）
それ自身 NP であって NP 困難である問題

ある問題が NP 困難であることを最初に証明することは難しい
計算機の動作のモデル化が必要→充足可能性判定問題が最初

それ以降は充足可能性判定問題をその問題に多項式時間還元可能であることを示せば良い

例：充足可能性判定→3-クリーク

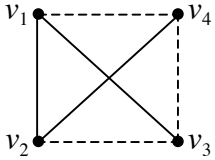
$$\left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee \overline{x_4}\right) \wedge \left(\overline{x_3} \vee x_4\right)$$



cf. 3-クリーク→充足可能性判定

頂点 v_i がクリークの j 番目の要素であることを x_i^j で表せば

$$\left(x_1^1 \vee x_2^1 \vee x_3^1 \vee x_4^1\right) \wedge \left(x_1^2 \vee x_2^2 \vee x_3^2 \vee x_4^2\right) \wedge \left(x_1^3 \vee x_2^3 \vee x_3^3 \vee x_4^3\right) \wedge$$
$$\left(\overline{x_1^1} \vee \overline{x_2^1}\right) \wedge \left(\overline{x_1^1} \vee \overline{x_3^1}\right) \wedge \left(\overline{x_1^1} \vee \overline{x_4^1}\right) \wedge \left(\overline{x_2^1} \vee \overline{x_3^1}\right) \wedge \left(\overline{x_2^1} \vee \overline{x_3^1}\right) \wedge \left(\overline{x_3^1} \vee \overline{x_4^1}\right) \wedge$$
$$\left(\overline{x_1^2} \vee \overline{x_2^2}\right) \wedge \left(\overline{x_1^2} \vee \overline{x_3^2}\right) \wedge \left(\overline{x_1^2} \vee \overline{x_4^2}\right) \wedge \left(\overline{x_2^2} \vee \overline{x_3^2}\right) \wedge \left(\overline{x_2^2} \vee \overline{x_3^2}\right) \wedge \left(\overline{x_3^2} \vee \overline{x_4^2}\right) \wedge$$
$$\left(\overline{x_1^3} \vee \overline{x_2^3}\right) \wedge \left(\overline{x_1^3} \vee \overline{x_3^3}\right) \wedge \left(\overline{x_1^3} \vee \overline{x_4^3}\right) \wedge \left(\overline{x_2^3} \vee \overline{x_3^1}\right) \wedge \left(\overline{x_2^3} \vee \overline{x_3^3}\right) \wedge \left(\overline{x_3^3} \vee \overline{x_4^3}\right) \wedge$$
$$\left(\overline{x_2^1} \vee \overline{x_3^2}\right) \wedge \left(\overline{x_2^1} \vee \overline{x_3^3}\right) \wedge \left(\overline{x_2^2} \vee \overline{x_3^1}\right) \wedge \left(\overline{x_2^2} \vee \overline{x_3^3}\right) \wedge \left(\overline{x_2^3} \vee \overline{x_3^1}\right) \wedge \left(\overline{x_2^3} \vee \overline{x_3^2}\right) \wedge$$
$$\left(\overline{x_3^1} \vee \overline{x_4^2}\right) \wedge \left(\overline{x_3^1} \vee \overline{x_4^3}\right) \wedge \left(\overline{x_3^2} \vee \overline{x_4^1}\right) \wedge \left(\overline{x_3^2} \vee \overline{x_4^3}\right) \wedge \left(\overline{x_3^3} \vee \overline{x_4^1}\right) \wedge \left(\overline{x_3^3} \vee \overline{x_4^2}\right) \wedge$$
$$\left(\overline{x_4^1} \vee \overline{x_1^2}\right) \wedge \left(\overline{x_4^1} \vee \overline{x_1^3}\right) \wedge \left(\overline{x_4^2} \vee \overline{x_1^1}\right) \wedge \left(\overline{x_4^2} \vee \overline{x_1^3}\right) \wedge \left(\overline{x_4^3} \vee \overline{x_1^1}\right) \wedge \left(\overline{x_4^3} \vee \overline{x_1^2}\right)$$



・ すべての NP 完全問題が同じくらい難しいのか？
「静的評価関数」が上手に作れる問題はそれほど難しくない

例：ナップザック問題

品物を分割できるのであれば $\frac{v_i}{w_i}$ の大きなものから順に重さの和が W を越えないところまで採用し、最後に品物を分割して W にちょうど収まるようにすればそれが Σv_i の最大値となることはすぐにわかる

ある品物を採用しなかったときの価値の最大値が次に大きな $\frac{v_i}{w_i}$ ×残りの重量で見積もることができる→絞り込み
→実用的には多項式時間で解ける
しかし、最悪の場合には全ての組み合わせを調べなければいけない

9.4 近似解法

例：Euclid 的巡回セールスマン問題

各辺の長さは 0 以上
任意の 3 点 A, B, C について $d_{AB}+d_{BC} \geqq d_{AC}$

1. ある点から始めて一番近い隣の点につないでいく	$O(n^2)$	$O(\log n)$倍
2. 短い辺から採用	$O(n^2 \log n)$	
3. 1 点から始めて最も近い点を加えていく	$O(n^2)$	2 倍
4. 最小木の往復	$O(n^2)$	2 倍

まず閉路を作ってから、交差しているところなどを適当に入れ替えてみて、短くなるようなら採用する

・ 遺伝的アルゴリズム

解候補を遺伝子で表現

良いものに絞り込む（淘汰）

遺伝子間で交叉・突然変異