プログラム設計とアルゴリズム
第2回 (10/4)

早稲田大学高等研究所　講師
福永津嵩

アルゴリズムとは何か？

- 「(数理的な)問題を解くための方法や手続き」

- 具体例：年齢当てゲーム

問題:
初対面のAさんの年齢を当てたいと考えます。
Aさんの年齢が20歳以上36歳未満であるとわかっているとします。

Aさんに「Yes/Noで答えられる質問」を4回まで出来るとした時、
あなたはこの年齢当てゲームで勝つ事が出来るでしょうか？

(前回の復習)効率的なアルゴリズム

- 候補を半分に絞るアルゴリズムを考える

図1.2

(前回の復習)効率的なアルゴリズム

- Q1: あなたは28歳未満ですか? →No　(28~35歳)

  Q2: あなたは32歳未満ですか? →No　(32~35歳)

  Q3: あなたは34歳未満ですか？→Yes (32~33歳)

Q4: あなたは33歳未満ですか? →No　(33歳)

あなたは33歳

(前回の復習)ランダウのO記法

- アルゴリズムの効率性を見積もる重要な基準として、**計算量**がある。
  計算機上での計算時間を大雑把に見積もる事が可能となる。

ランダウのO記法

(前回の復習)ランダウのO記法

- $T(N) = 3N^2 + 5N + 100$ を再考する。まず$T(N)$をNで割ってみると、

$$3N + 5 + \frac{100}{N}$$

となり、明らかに定数cで抑える事は出来ない。

よって　$T(N) = O(N)$ ではない。

- 次に$T(N)$をN^2で割ると、

$$3 + \frac{5}{N} + \frac{100}{N^2}$$

となり、Nが十分に大きい時、これはc=4で抑えられる。

よって、 $T(N) = O(N^2)$

第三章

設計技法(1):全探索

設計技法(1) 全探索

- 全探索：考えられる可能性を全て列挙し、その中から条件を満たす解があるかを探し出すアルゴリズム

- 原理的には多くの問題を全探索で解くことが可能だが、問題によっては現実的な時間では可能性を全て列挙することは出来ないこともある(将棋の盤面の列挙など)。

- しかし、小さい問題については全探索が可能であることも多いほか、より効率的なアルゴリズムを設計する上での基礎となることも多い。

全探索(1)：線形探索法

問題:
N個の整数 a_i (i = 0, 1, … N-1)が与えられた時、
a_i =vとなる整数値があるかを判定しなさい。

- 線形探索では、1つ1つの要素を順番に調べていき、合致するデータが存在するかどうかを判定する。

- たとえば、a= (4,3,12,7,11)という数列の中に、v=7が含まれているかどうかを判定する方法は次の通り。

全探索(1)：線形探索法

図3.1

線形探索法のソースコード

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // 入力を受け取る
    int N, v;
    cin >> N >> v;
    vector<int> a(N);
    for (int i = 0; i < N; ++i) cin >> a[i];

    // 線形探索
    bool exist = false;   // 初期値は false に
    for (int i = 0; i < N; ++i) {
        if (a[i] == v) {
            exist = true;  // 見つかったらフラグを立てる
        }
    }

    // 結果出力
    if (exist) cout << "Yes" << endl;
    else cout << "No" << endl;
}
```

| boolはtrueかflaseを取る型

線形探索法の計算量

- for文はN回ループが行われるので、計算量は*O(N)*

- 16行目のexist=true; の後に、break;と書くことで
  for文を抜け出す事ができる。これで計算が早く終了する可能性がある。
  (存在することがわかった場合残りを探索する必要はない)

- しかし、aの中にvが存在しない場合はやはりN回ループを回す必要がある
  よって、break文を入れても(最悪時間)計算量に変化はない。

線形探索法の応用

- プログラムを少し工夫することで、$a_i=v$を満たすiも知る事が出来る

```
// 線形探索
int found_id = -1; // 初期値は -1 などありえない値に
for (int i = 0; i < N; ++i) {
    if (a[i] == v) {
        found_id = i; // 見つかったら添字を記録
        break; // ループを抜ける
    }
}
```

線形探索法の応用

- また線形探索法を活用することで、aの中の最小値(や最大値)を発見することもできる。

```cpp
#include <iostream>
#include <vector>
using namespace std;
const int INF = 20000000;   // 十分大きな値に

int main() {
    // 入力を受け取る
    int N;
    cin >> N;
    vector<int> a(N);
    for (int i = 0; i < N; ++i) cin >> a[i];

    // 線形探索
    int min_value = INF;
    for (int i = 0; i < N; ++i) {
        if (a[i] < min_value) min_value = a[i];
    }

    // 結果出力
    cout << min_value << endl;
}
```

全探索(2)：ペアの全探索

問題:
N個の整数 a_i (i = 0, 1, … N-1)と、N個の整数b_i (i = 0, 1, … N-1)
が与えられ、２個の整数列から1つずつ整数を選んで和を計算する
ものとする。その和の値のうち、K以上の範囲での最小値を求めなさい。

- N=3, K=10, a= (8, 5, 4), b = (4, 1, 9)なら
  8+4 =12 が最小解となる。

- 全解探索では、考えられる場合の数はN^2通り全てを探索する。

全探索(2)：ペアの全探索

```cpp
#include <iostream>
#include <vector>
using namespace std;
const int INF = 20000000;   // 十分大きな値に

int main() {
    // 入力を受け取る
    int N, K;
    cin >> N >> K;
    vector<int> a(N), b(N);
    for (int i = 0; i < N; ++i) cin >> a[i];
    for (int i = 0; i < N; ++i) cin >> b[i];
```

全探索(2)：ペアの全探索

```
14      // 線形探索
15      int min_value = INF;
16      for (int i = 0; i < N; ++i) {
17          for (int j = 0; j < N; ++j) {
18              // 和が K 未満の場合はスキップ
19              if (a[i] + b[j] < K) continue;
20
21              // 最小値を更新
22              if (a[i] + b[j] < min_value) {
23                  min_value = a[i] + b[j];
24              }
25          }
26      }
27
28      // 結果出力
29      cout << min_value << endl;
```

全探索(3)：組み合わせの全探索

問題(部分和問題):
N個の整数 a_i (i = 0, 1, … N-1)と正の整数Wが与えられる。この時、整数列aの中から値を何個か選び、その総和をWとする事が出来るかどうかを判定しなさい。

- N = 5, W = 10とする。
  a = {1,2,4,5,11}であれば、1+4+5 = 10なのでYes
  a = {1,3,5,8,11}であれば、10を作ることはできないのでNo

- 整数の組み合わせを全て調べることで部分和問題を解く
  1つの整数について着目すると、その整数を選ぶか選ばないかの２通り
  よって、整数がN個あると、その組み合わせの総数は2^N通り

全探索(3)：組み合わせの全探索

- 整数の二進数表現とビット演算で全探索を行う(bit全探索)

- 組み合わせ(部分集合)を次のように二進数で表現する。

表3.1

- 二進法でN桁あれば、整数の組み合わせを全て調べる事が出来る

C++における整数の表現

- C++において、整数は二進法で表現されており、
  一般にはint型整数は4バイト(=32ビット)である。

- 一般には、最上位のビットが0であれば正の数、1であれば
  負の数となり、残り31ビットが二進法での数値表現となる。
  (負の数は2の補数表現などが利用されるが、説明は省略)

- よってint型の変数は、整数の部分集合とみなす事が出来る

  (int x=3は、a_0とa_1の組み合わせを表しているとみなせる)

ビット演算

・ C++における、二進法表記したときのbit演算を行う演算子
　ここでは、二進数表記は頭に0bとつけるものとする。

1. &演算子　(各桁ごとにANDを取る)
　例) 6 (= 0b110) & 3 (= 0b011) = 2 (= 0b010)

2. |演算子　(各桁ごとにORを取る)
　例) 6 (= 0b110) | 3 (= 0b011) = 7 (= 0b111)

3. >>演算子または<<演算子　(ビット列を右または左にずらす)
　例) 3 << 2 (数字の3をビット列で見たときに、左に２個ずらす)
　3 (= 0b00011)を左に2個ずらすと、12(= 0b01100)

部分集合とbit演算

- 整数で表現された部分集合にbit演算を行うことで、
  その部分集合にi番目の要素が含まれるかを判定出来る

code 3.5

- 上記のコードは、C++では0以外の整数値はtrueを表し、
  0はfalseを表すことを利用している。

部分集合とbit演算

- 例)

  N=8として、部分集合{a_0,a_2,a_3,a_6}を考える。

  これはbit表現では、01001101となる。

表3.2

部分和問題へのbit全探索アルゴリズム

問題(部分和問題):
N個の整数 a_i (i = 0, 1, … N-1)と正の整数Wが与えられる。この時、整数列aの中から値を何個か選び、その総和をWとする事が出来るかどうかを判定しなさい。

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // 入力受け取り
    int N, W;
    cin >> N >> W;
    vector<int> a(N);
    for (int i = 0; i < N; ++i) cin >> a[i];
```

部分和問題へのbit全探索アルゴリズム

```
12      // bit は 2^N 通りの部分集合全体を動く
13      bool exist = false;
14      for (int bit = 0; bit < (1 << N); ++bit)
15      {
16          int sum = 0; // 部分集合に含まれる要素の和
17          for (int i = 0; i < N; ++i) {
18              // i 番目の要素 a[i] が部分集合に含まれているかどうか
19              if (bit & (1 << i)) {
20                  sum += a[i];
21              }
22          }
23
24          // sum が W に一致するかどうか
25          if (sum == W) exist = true;
26      }
27
28      if (exist) cout << "Yes" << endl;
29      else cout << "No" << endl;
```

- 計算量を考えると、外側のfor文は2^N、内側のfor文はN回ループする。
  シフト演算は定数時間で行う事が出来るため、計算量は $O(N2^N)$

要点の再確認

- 0と1からなるビット列は、二進法によって整数とみなすことができる。
  実際に計算機内部では、int型など、この二進法を利用して整数を
  表現している。

- また、0と1からなるビット列は、整数の組み合わせを表現すること
  もできる。

- よって、int型の整数は、整数の組み合わせを表現しているとと
  みなしてプログラムを書く事ができる。

第四章

設計技法(2):再帰と分割統治法

再帰呼び出し

- ある関数において、自分自身を呼び出すことを再帰呼ぶ出しという。

```
1  int func(int N) {
2      if (N == 0) return 0;
3      return N + func(N - 1);
4  }
```

図4.1

再帰の例(1):ユークリッドの互除法

- ２つの整数m, nの最大公約数をGCD(m, n)とする
  (Greatest Common Divisor)

- このとき、m % n = rとすると、
  GCD(m, n) = GCD(n, r)が成立する。

- たとえばm＝51、n=15とすると、rは6であり、
  GCD(51, 15) = GCD(15, 6)が成立する。

再帰の例(1):ユークリッドの互除法

- 最大公約数の性質を繰り返し適用することで、最終的に
  最大公約数を得る方法をユークリッドの互除法という。

教科書46P中部

再帰の例(1):ユークリッドの互除法

- 再起を用いてユークリッドの互除法を書くと、次のようになる

```cpp
#include <iostream>
using namespace std;

int GCD(int m, int n) {
    // ベースケース
    if (n == 0) return m;

    // 再帰呼び出し
    return GCD(n, m % n);
}

int main() {
    cout << GCD(51, 15) << endl; // 3 が出力される
    cout << GCD(15, 51) << endl; // 3 が出力される
}
```

再帰の例(2):フィボナッチ数列

- 次の要件を満たす数列をフィボナッチ数列という

1. $f_0 = 0$
2. $f_1 = 1$
3. $f_n = f_{n-1} + f_{n-2}$ $(n > 2)$

- すなわち、0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …となる。

再帰の例(2):フィボナッチ数列

- フィボナッチ数列は、再帰呼び出しを2回行うことで計算できる

```
1   int fibo(int N) {
2       // ベースケース
3       if (N == 0) return 0;
4       else if (N == 1) return 1;
5
6       // 再帰呼び出し
7       return fibo(N - 1) + fibo(N - 2);
8   }
```

計算順序

図4.2

再帰の例(2):フィボナッチ数列

- 一方で、このコードは非常に無駄が多い

図4.3

- 計算量としては指数時間アルゴリズムとなる

再帰の例(2):フィボナッチ数列

- for文でフィボナッチ数列を求める場合、*O(N)*の計算量で求められる

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main() {
6       vector<long long> F(50);
7       F[0] = 0, F[1] = 1;
8       for (int N = 2; N < 50; ++N) {
9           F[N] = F[N - 1] + F[N - 2];
10          cout << N << " ¹àÌÜ: " << F[N] << endl;
11      }
12  }
```

再帰の例(2):フィボナッチ数列

- 同じ引数に対しては計算を2回は行わない**メモ化**をすることで、
  再帰呼び出しでも$O(N)$の計算量でフィボナッチ数列を得ることができる

```
5   // fibo(N) の答えをメモ化する配列
6   vector<long long> memo;
7
8   long long fibo(int N) {
9       // ベースケース
10      if (N == 0) return 0;
11      else if (N == 1) return 1;
12
13      // メモをチェック（すでに計算済みならば答えをリターンする）
14      if (memo[N] != -1) return memo[N];
15
16      // 答えをメモ化しながら，再帰呼び出し
17      return memo[N] = fibo(N - 1) + fibo(N - 2);
18  }
```

再帰の例(2):フィボナッチ数列

- 同じ引数に対しては計算を2回は行わない**メモ化**をすることで、
  再帰呼び出しでも*O(N)*の計算量でフィボナッチ数列を得ることができる

```
20   int main() {
21       // メモ化用配列を -1 で初期化する
22       memo.assign(50, -1);
23
24       // fibo(49) をよびだす
25       fibo(49);
26
27       // memo[0], ..., memo[49] に答えが格納されている
28       for (int N = 2; N < 50; ++N) {
29           cout << N << " 項目: " << memo[N] << endl;
30       }
31   }
```

再帰の例(3):再帰関数を用いる全探索

問題(部分和問題):
N個の整数 a_i (i = 0, 1, … N-1)と正の整数Wが与えられる。この時、整数列aの中から値を何個か選び、その総和をWとする事が出来るかどうかを判定しなさい。

- 再帰関数を利用する方法では、以下の2つの方法を場合分けして考える。

  - a_{N-1} を選ばないとき
  - a_{N-1} を選ぶとき

再帰の例(3):再帰関数を用いる全探索

図4.4

再帰の例(3):再帰関数を用いる全探索

- N=4, a = (3, 2, 6, 5), W = 14であるとした場合の探索例

図4.5

再帰の例(3):再帰関数を用いる全探索

```
22  int main() {
23      // 入力
24      int N, W;
25      cin >> N >> W;
26      vector<int> a(N);
27      for (int i = 0; i < N; ++i) cin >> a[i];
28
29      // 再帰的に解く
30      if (func(N, W, a)) cout << "Yes" << endl;
31      else cout << "No" << endl;
32  }
```

再帰の例(3):再帰関数を用いる全探索

```
5   bool func(int i, int w, const vector<int> &a) {
6       // ベースケース
7       if (i == 0) {
8           if (w == 0) return true;
9           else return false;
10      }
11
12      // a[i - 1] を選ばない場合
13      if (func(i - 1, w, a)) return true;
14
15      // a[i - 1] をぶ場合
16      if (func(i - 1, w - a[i - 1], a)) return true;
17
18      // どちらも false の場合は false
19      return false;
20  }
```

再帰の例(3):再帰関数を用いる全探索

- 計算量は再帰関数の呼び出し回数(再帰関数の中身は定数時間なので)
  つまり、前に紹介した図における矢印の数

- これは、

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

- これはまだ指数時間アルゴリズムだが、
  bit全探索を活用した$O(N2^N)$に比べると計算量が小さくなっている。

再帰の例(3):再帰関数を用いる全探索

- メモ化を行うことで、計算量を*O(NW)*と多項式時間アルゴリズムにすることができる。

- func(i, w, a)の結果をメモする二次元配列memo[][]を作成する。
  memo[i][w] = func(i, w, a)の結果が入るようにする。

- memo[i][w]が既に計算済みであるなら、それ以上の再帰を打ち切る。
  memoはN×Wの二次元配列であるから、最悪計算量は*O(NW)*となる。

- (実装は各自考えてみてください。講義で示したコードを上記のような方針で変更すれば良いです。)

分割統治法

- 部分和問題で見たように、与えられた問題を部分的な小さな問題に分割して、それらの問題を再帰的に解き、その結果を組み合わせて元の問題の解を得る方法を分割統治法と呼ぶ。

まとめ

- 問題を解くための設計技法として、全探索と再帰を紹介した。

- 全探索はありえる解を全てしらみつぶしに探索する手法である。
  組み合わせの全探索を効率的にプログラミングする手法として、
  bit全探索と呼ばれる手法がある。全探索は効率が悪い事も多いが、
  アルゴリズム設計の基礎として重要である。

- 再帰は、自分自身を呼び出す関数手続きのことである。
  メモ化によって再帰を高速化できることがある。