プログラム設計とアルゴリズム
第4回 (10/18)

早稲田大学高等研究所　講師
福永津嵩

(前回の復習)動的計画法の例題

Frog問題:

N個の足場があって、その高さが h_i ($i = 0, 1, \cdots N-1$)で与えられている。

カエルは次のどちらかの行動で移動していく。

- 足場iから足場i+1へ、$|h_i - h_{i+1}|$のコストで移動する。
- 足場iから足場i+2へ、$|h_i - h_{i+2}|$のコストで移動する。

足場0から足場N−1へカエルが移動する時、

コストの総和の最小値を求めなさい。

図5.1

(前回の復習) 動的計画法の例題

- N=7, h=(2,9,4,5,1,6,10)とする。
  この時、足場0から足場1へのコストは|2-9|=7であり、
  足場0から足場2へのコストは|2-4|=2である。

- 足場を「丸(頂点)」、足場間の移動を「矢印(辺)」、移動にかかるコストを「重み」として表現すると、次のように表せる(このような表現をグラフと呼ぶ)

図5.2

(前回の復習) 動的計画法の例題

図5.6

(前回の復習) ナップサック問題

問題:
N個の品物があり、i番目の品物の重さがw_i、価値がv_iであるとする。
重さの総和がWを超えないように商品を選んだ時、価値の総和として考えられる最大値を求めなさい。(Wやw_iは整数とする。)

- 例: 6個の品物があり、W= 7、
  (w, v) ={(2, 3), (1, 2), (3, 6), (2, 1), (1, 3), (5, 85)}とする。

- 1~4番目を選ぶとすると、重さの総和は 2+1+3+2 > 7より違反する。

- 6番目だけ選ぶとすると、重さの総和は 5 <= 7で条件を満たし、
  価値の総和は85となるが、もう少し増やせそう

(前回の復習) ナップサック問題

- dp[i][w]を、最初のi個の品物のうち、重さがwを超えないように選んだ時の価値の総和の最大値とする。

図5.10

(前回の復習) 編集距離の計算

編集距離

2 つの文字列 S, T が与えられます．S に以下の 3 通りの操作を繰り返し施すことで T に変換したいものとします．そのような一連の操作のうち，操作回数の最小値を求めてください．なお，この最小値を S と T との編集距離とよびます．

- 変更：S 中の文字を 1 つ選んで任意の文字に変更する
- 削除：S 中の文字を 1 つ選んで削除する
- 挿入：S の好きな箇所に好きな文字を 1 文字挿入する

- HANABIとHNABIAでは、

  HANABI－挿入
  |X||||X
  H－NABIA
  削除

  のため、編集距離は2となる。

動的計画法の例(2):編集距離

図5.12

第六章

設計技法(4):二分探索法

二分探索法

- 探索範囲を半減させていくことで解を発見する手法

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？

配列の二分探索

- ソート済みの配列が与えられた時、配列内にある値が存在しているかを調べる。

- 例)
  N = 8のソート済み配列 a = {3, 5, 8, 10, 14, 17, 21, 39}
  の中にkey = 9が含まれているかを検索する。

二分探索では次のように検索を行う。

まず、left =0, right = N − 1 = 7として初期化する。その後、

- key = a[(left + right) / 2] ならば，“Yes” を返して探索を終了します
- key < a[(left + right) / 2] ならば，配列の左半分のみを残します
- key > a[(left + right) / 2] ならば，配列の右半分のみを残します

配列の二分探索

図6.1

配列の二分探索

- 実行順:
  left = 0、 right = 7、この時、mid = (0+7)/2 = 3
  key = 9 < a[3] =10より、配列の左半分を残す
  (もし9があるとしたら左側にしかないということ)

- 具体的には、right = 3−1=2として、同じ操作を繰り返す。
  left = 0、 right = 2、mid = (0+2)/2 = 1
  key = 9 > a[1] = 5より、配列の右半分を残す。(left = 1+1= 2)

- left = 2、right = 2, mid = (2+2)/2 = 2
  key = 9 > a[2] = 8より、配列の右半分を残す。(left = 2+1 = 3)

  left > rightになってしまったので処理終了。aにkeyが存在しない。

配列の二分探索

```
8   // 目的の値 key の添字を返す（存在しない場合は -1）
9   int binary_search(int key) {
10      int left = 0, right = (int)a.size() - 1; // 配列 a の左端と右端
11      while (right >= left) {
12          int mid = left + (right - left) / 2; // 区間の真ん中
13          if (a[mid] == key) return mid;
14          else if (a[mid] > key) right = mid - 1;
15          else if (a[mid] < key) left = mid + 1;
16      }
17      return -1;
18  }
```

- 1ステップで配列サイズが半減するので、計算量は$O(\log N)$となる。

C++のstd::lower_bound()

- C++のstd::lower_bound()関数は、ソート済み配列aが与えられた時
  a[i] >= keyとなる最小の添字i (正確にはイテレータ)を返す関数である。

- これは先ほどの問題とは
  1. aの中にkeyがなくても、keyより大きい最小値がわかる
  2. aの中にkeyが複数あった時に、その最小の添え字がわかる。
  という点で異なる。

- これも二分探索法をより一般化することでO(log N)で解く事が出来る。

一般化した二分探索法

一般化した二分探索法

一般化した二分探索法

図6.4

図6.5

さらに一般化した二分探索法

- falseとtrueに完全に二分されているという仮定(単調性)を取り除く

一般化した実数上の二分探索法

図6.7

二分探索法の例(1): ペア和のK以上の中での最小値

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。

- 以前全探索で扱った問題を再考する。

  全探索ではO(N^2)であったが、二分探索を活用することで
  実はO(N log N)で解く事が出来る。

二分探索法の例(1): ペア和のK以上の中での最小値

- まず、bをソートする。
  第7回で講義するが、この計算量はO(N log N)となる。

- あるaᵢが固定されているとして考えると、問題は次のようになる。

> 問題:
> N個の整数b_i (i = 0, 1, … N−1)が与えられる。そのうち、
> K−a_i以上の範囲での最小値を求めなさい。

これは、lower_bound()関数を用いる事でO(log N)で解く事が出来る。

- aの値はN個あるので、計算量は全部でO(N log N)となる。

二分探索法の例(1): ペア和のK以上の中での最小値

```
    // 暫定最小値を格納する変数
    int min_value = INF;

    // b をソート
    sort(b.begin(), b.end());

    // b に無限大を表す値（INF）を追加しておく
    // これを行うことで、iter = b.end() となる可能性を除外する
    b.push_back(INF);

    // a を固定して解く
    for (int i = 0; i < N; ++i) {
        // b の中で K - a[i] 以上の範囲での最小値を示すイテレータ
        auto iter = lower_bound(b.begin(), b.end(), K - a[i]);

        // イテレータの示す値を取り出す
        int val = *iter;

        // min_value と比較する
        if (a[i] + val < min_value) {
            min_value = a[i] + val;
        }
    }
    cout << min_value << endl;
}
```

二分探索法の例(2): 射撃王

射撃王問題

二分探索法の例(2): 射撃王

- 二分探索法で解くことを考える。

  left = 0とし、
  right = max $(H_0 + (N-1)S_0, \cdots H_{N-1} + (N-1)S_{N-1})$とする。

  ここでM = right−leftとする。そしてleftとrightからmidを計算し、
  midのペナルティ内で全ての風船を割る事が出来るかを判定する。

- 出来なければleft = midとし、出来ればright = midとすることで、
  midを再計算する。同様の手続きを繰り返す。

  二分探索に要する計算時間は O(logM)

二分探索法の例(2): 射撃王

- よって「整数xが与えられた時に、最終的なペナルティをx以下に出来るか」を解けば良い。

- 各風船のペナルティをx以下に抑えるために、各風船を何秒以内に割れば良いかがわかる。

- 時間的に余裕のない風船から割っていき、全ての風船を割る事が出来るかを考えれば良い。

- ソートを必要とするため、個別問題の計算量はO(NlogN)となる。二分探索を組み合わせると、全体の計算量はO(NlogNlogM)となる。

二分探索法の例(2): 射撃王

```
// 二分探索
long long left = 0, right = INF;
while (right - left > 1) {
    long long mid = (left + right) / 2;

    // 判定
    bool ok = true;
    vector<long long> t(N, 0); // 各風船を割るまでの制限時間
    for (int i = 0; i < N; ++i) {
        // そもそも mid が初期高度より低かったらfalse
        if (mid < h[i]) ok = false;
        else t[i] = (mid - h[i]) / s[i];
    }
    // 時間制限がさし迫っている順にソート
    sort(t.begin(), t.end());
    for (int i = 0; i < N; ++i) {
        if (t[i] < i) ok = false; // 時間切れ発生
    }

    if (ok) right = mid;
    else left = mid;
}
```

第七章

設計技法(5):貪欲法

貪欲法

- 今後のことは考えず、目の前にある選択肢の中から最も良さそうなものを選択していくことを繰り返す手法を貪欲法と呼ぶ。

例題:
500円玉、100円玉、50円玉、10円玉、5円玉、1円玉がそれぞれa_0、a_1、a_2、a_3、a_4、a_5枚あったとする。X円を支払う時、最も支払うコインの枚数が少なくなるような組み合わせを求めよ。

- この問題に対する貪欲法は、最も額が大きいコインから払えるだけ払っていくことである。例えばX = 1299円で各コインは十分にあるとすると、
1299 = 500*2 + 100*2 + 50*1 + 10*4 + 5*1 + 1*4となり、14枚となる。
実際にこれが最適である。

貪欲法

```
5   // コインの金額
6   const vector<int> value = {500, 100, 50, 10, 5, 1};
```

```
15      // 貪欲法
16      int result = 0;
17      for (int i = 0; i < 6; ++i) {
18          // 枚数制限がない場合の枚数
19          int add = X / value[i];
20
21          // 枚数制限を考慮
22          if (add > a[i]) add = a[i];
23
24          // 残り金額を求めて，答えに枚数を加算する
25          X -= value[i] * add;
26          result += add;
27      }
28      cout << result << endl;
```

貪欲法の例(1):区間スケジューリング問題

問題:
N個の仕事があり、各仕事は時刻s_iに始まりt_iに終わる。この時、最大何個の仕事をする事が出来るかを求めなさい。

図7.2

貪欲法の例(1):区間スケジューリング問題

- 「開始時間が早い順番に仕事を選んでいく」という貪欲法が考えられるが、「最初に始まるが最後に終わる仕事」などを考えると明らかに正しい方法ではない。

- 「終了時間が早い順番に仕事を選んでいく」という貪欲法が適切。

すなわち、

1. 終了時刻に基づいて仕事をソートする。
2. 一番終了時間の速い仕事を選ぶ。その仕事と重なっている他の仕事は受けられないので削除する。
3. 残っているうち、一番終了時間の速い仕事を選ぶ。これを繰り返す。

貪欲法の例(1):区間スケジューリング問題

図7.5

貪欲法の例(1):区間スケジューリング問題

- この場合の貪欲法は次の理由により最適である。

1. 仕事数を最も多くする仕事の選び方Xが、終了時刻の最も速い仕事p
   を含んでいなかったとする。

2. この時、Xはpと重なっている仕事qを行なっている。
   ここで、q以外の仕事はpと重なっていないため、qの代わりにp
   をしても仕事の個数は変化しない。よって、仕事数を最も多くする
   仕事の組み合わせに、pを含めて問題ない。

3. この手続きを繰り返す。

- 計算量はソートにかかる時間でO(NlogN)となる。

貪欲法の例(1):区間スケジューリング問題

```
7   // 区間を pair<int,int> で表す
8   typedef pair<int,int> Interval;
9
10  // 区間を終端時刻で大小比較する関数
11  bool cmp(const Interval &a, const Interval &b) {
12      return a.second < b.second;
13  }
```

```
23      // 終端時刻が早い順にソートする
24      sort(inter.begin(), inter.end(), cmp);
25
26      // 貪欲に選ぶ
27      int res = 0;
28      int current_end_time = 0;
29      for (int i = 0; i < N; ++i) {
30          // 最後に選んだ区間と被るのは除く
31          if (inter[i].first < current_end_time) continue;
32
33          ++res;
34          current_end_time = inter[i].second;
35      }
36      cout << res << endl;
```

貪欲法の例(2):Multiple Array

Multiple Array問題

貪欲法の例(2):Multiple Array

- ボタンiを押す回数をD_iとすると、要件は

  - $A_0 + (D_0 + D_1 + \cdots + D_{N-1})$ は B_0 の倍数
  - $A_1 + (D_1 + \cdots + D_{N-1})$ は B_1 の倍数
  - $\vdots$
  - $A_{N-1} + D_{N-1}$ は B_{N-1} の倍数

すなわち、iが大きい方から順番に倍数になるよう決めていけば良い。

貪欲法の例(2):Multiple Array

```
12      // 答え
13      long long sum = 0;
14      for (int i = N - 1; i >= 0; --i) {
15          A[i] += sum; // 前回までの操作回数を足す
16          long long amari = A[i] % B[i];
17          long long D = 0;
18          if (amari != 0) D = B[i] - amari;
19          sum += D;
20      }
21      cout << sum << endl;
```

(教科書になし)

設計技法(6):乱択アルゴリズム

乱数と乱択アルゴリズム

- 乱拓アルゴリズムとは、乱数を利用して確率的な挙動を
  示すアルゴリズムのことである。特に、正しい解(あるいはその近似解)
  を高い確率で得られるが、間違える可能性があるアルゴリズムを
  モンテカルロ法と呼ぶ。

- 乱数列とは、乱数の列のことであり、平たく言えば
  次に来る数字がランダムであり全く予測できない数列の事である。

- しかしコンピュータは、決定的な計算だけしか行う事ができないので
  全く規則のない数列を作る事はできない。よって真の乱数列は作れない。
  そのため、擬似乱数列を作って乱数とみなしている。

線形合同法

- $x_{i+1} = ax_i + b \pmod p$とする漸化式によって乱数列を生成する。
  (a, b, pは与えられている)
  N個乱数を得たければ、この漸化式をN回計算すればよい。

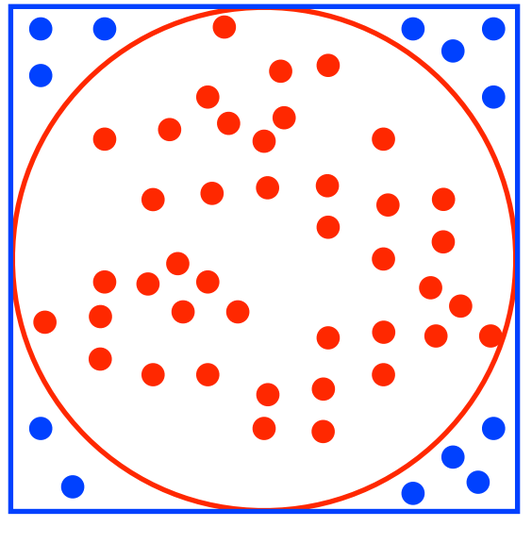
- x_0が同じであれば、必ず同じ乱数列になってしまうので、実行のたびにx_0を変えてあげる必要がある。このようなx_0をseed値と呼ぶ。
  (再現性確保のために同じ結果を得たい場合には、seed値を保存しておく)
  seed値としては、現在時刻や計算機の状態に基づいて計算された値が利用される。

- 線形合同法は規則性が高いという問題があり、現在では
  メルセンヌ・ツイスター法などといったより高度な手法が使われる。

C++11における乱数生成とその活用

```cpp
#include <random> //乱数ライブラリを使用するときにincludeする。
#include <iostream>
#include <vector>
using namespace std;

int main(void){
  random_device rd;
  //現在時刻やハードウェアの状態を利用して乱数を生成。これだけでも良いが
  //生成に時間がかかるため、この結果をseed値として利用して、より高速な
  //乱数生成アルゴリズムを使うのが普通。

  mt19937 mt(rd()); //メルセンヌツイスター法
  for(int i = 0; i < 5; i++){
    cout << mt() << " ";
  }
  cout << endl;

  normal_distribution<> dist(0.0, 5.0); //平均0.0、分散5.0の正規分布
  for(int i = 0; i < 5; i++){
    cout << dist(mt) << " ";
  }
  cout << endl;

  vector<int> v(5,0);
  for(int i = 0; i < 5; i++){
    v[i] = i;
  }
  shuffle(v.begin(), v.end(), mt); //シャッフル
  for(int i = 0; i < 5; i++){
    cout << v[i] << " ";
  }
  cout << endl;
}
```

乱択アルゴリズムの例(1):円周率の計算



- [-1，1]から乱数を2つ取り、
  それをある点のx座標、y座標とみなす。
  このような点をN個取る。

- N個の点のうち、原点を中心とする半径が1の円
  の中に何個の点が入っているのかを調べる。(原点との距離が
  1以下であるかを調べる。) 入っていた個数をM個とする。

- 面積比を考えると、Nが十分に大きい時にM/N ＝ $\pi/4$に従う。
  この事で円周率を計算できる。

乱択アルゴリズムの例(2):行列積の計算

- N*Nの行列A, B, Cがあった時に、AB = Cであるかを判定したい。
  普通に行列積を計算した場合、$O(N^3)$かかるが、乱拓アルゴリズムによって、$O(N^2)$で計算する事が可能である。

- 要素がランダムに0または1である、長さNのベクトルxを考える。
  そしてABxとCxを計算し、これが同じ値になるかを考える。
  ABx = A(Bx)より、この計算は$O(N^2)$で行う事が出来る。

- AB = Cの時は、同じ答えになる。
  AB ≠ Cの時は、高々1/2の確率で同じ答えになる。
  少なくとも、違う答えであれば、AB ≠ Cである。

乱択アルゴリズムの例(2):行列積の計算

- xを変えて同じ計算を繰り返す。M回計算を行ったとすると、
  AB ≠ Cであるにも関わらず全て同じ答えになる確率は、高々$1/2^M$回
  なので、非常に高い確率で正しく判定する事が可能である。

- AB ≠ Cの時、同じになる確率が高々1/2であることの証明

1. D = AB-Cとすると、Dは非零行列で$Dx = 0$
2. Dのうち非零成分を持つ行dを取り出し、その非零成分をd_iとする。
   $d \cdot x = 0$より、$d_i x_i = -\sum_{j \neq i} d_j x_j$
3. x_i以外が先に決まっていたとして、x_iがこの数式を満たすためには
   その解が0または1であり、かつ乱数によってその解がx_iとして選ば
   れる事が必要。よって高々1/2

まとめ

- 問題を解くための設計技法として、二分探索法・貪欲法・
  乱択アルゴリズムを紹介した。

- 二分探索法とは、探索範囲を半減させながら解を発見する方法である。

- 貪欲法とは、目の前にある選択肢から最も良さそうなものを
  選択する方法である。問題の構造によっては、正解を選ぶ保証はない。

- 乱択アルゴリズムとは、乱数を用いることで解を得る手法であり、
  (間違っている可能性もあるが)高速に解を得られる事がある。