プログラム設計とアルゴリズム
第3回 (10/11)

早稲田大学高等研究所　講師
福永津嵩

(前回の復習)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

- 具体例

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。

(前回の復習)全探索

```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;
}
```

(前回の復習)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。

- 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)ことで、プログラムを効率的に実装することが可能である。

表3.1

(前回の復習)組み合わせの全探索

```
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;
```

(前回の復習)再帰

- 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

(前回の復習)再帰

- 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

(前回の復習)再帰

- 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

(前回の復習) メモ化再帰

- 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  }
```

第五章

設計技法(3):動的計画法

動的計画法とは何か？

- 与えられた問題を部分問題に分解し、
  各部分問題に関する解をメモ化しながら、
  部分問題の解を統合することで元の問題を解く手法。

- (これだけでは何もわからないですね)

- 実のところ、前回紹介した、部分和問題を再帰で解く際に
  メモ化を行う手法は、定義的に見てこの動的計画法であった。

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

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

動的計画法の例題

問題の言い換え:
先ほどのグラフにおいて、頂点0から頂点6まで辺をたどる方法のうち、各辺の重みの総和の最小値を求めなさい。

- 頂点6への最小コストはいきなりはわからないので、まずは頂点1や2などへの最小コストを考える(部分問題に分解)

- 頂点iへの最小コストをdp[i]にメモ化することとする。
  まず初期位置のコストは0なので、dp[0] = 0

図5.3

動的計画法の例題

- 頂点1へは、頂点0から行くしかない。よって、dp[1]=dp[0]+7=7

図5.4

- 頂点2へは、頂点0または頂点1から行くことができる。
  頂点0からは、dp[0]+2=2
  頂点1からは、do[1]+5=12のコストがかかる。
  このうちコストの小さい方をメモするので、dp[2]=2

- この手続きを繰り返す

動的計画法の例題

図5.6

動的計画法の例題

```cpp
#include <iostream>
#include <vector>
using namespace std;
const long long INF = 1LL << 60; // 十分大きい値とする（ここでは 2^60）

int main() {
    // 入力
    int N; cin >> N;
    vector<long long> h(N);
    for (int i = 0; i < N; ++i) cin >> h[i];

    // 配列 dp を定義（配列全体を無限大を表す値に初期化）
    vector<long long> dp(N, INF);

    // 初期条件
    dp[0] = 0;

    // ループ
    for (int i = 1; i < N; ++i) {
        if (i == 1) dp[i] = abs(h[i] - h[i - 1]);
        else dp[i] = min(dp[i - 1] + abs(h[i] - h[i - 1]),
                         dp[i - 2] + abs(h[i] - h[i - 2]));
    }

    // 答え
    cout << dp[N - 1] << endl;
}
```

- 計算量は*O(N)*

動的計画法における緩和

- dp配列の値を小さいほうへと更新していく操作
  緩和の考えのもとfrog問題を捉えると次のようになる

図5.7

chmin関数

- 緩和処理を行うための関数　(choose min)

```
1   template<class T> void chmin(T& a, T b) {
2       if (a > b) {
3           a = b;
4       }
5   }
```

  様々な型(int, long, doubleなど)に対して統一的に関数を記述するために
  C++ではテンプレート関数と呼ばれるものが用意されている。

- また、T&の&は参照を意味しており、C++の機能
  chmin関数での計算の結果が、chmin関数を抜けた後も保存される
  (やりたいことはCでのポインタ引数と同じ)

緩和を意識したflog問題の解法

```cpp
const long long INF = 1LL << 60; // 十分大きい値とする（ここでは 2^60）

int main() {
    // 入力
    int N; cin >> N;
    vector<long long> h(N);
    for (int i = 0; i < N; ++i) cin >> h[i];

    // 初期化（最小化問題なので INF に初期化）
    vector<long long> dp(N, INF);

    // 初期条件
    dp[0] = 0;

    // ループ
    for (int i = 1; i < N; ++i) {
        chmin(dp[i], dp[i - 1] + abs(h[i] - h[i - 1]));
        if (i > 1) {
            chmin(dp[i], dp[i - 2] + abs(h[i] - h[i - 2]));
        }
    }

    // 答え
    cout << dp[N - 1] << endl;
}
```

貰う遷移形式と配る遷移形式

- 頂点iに対して向かってくる遷移を「貰う遷移形式」
  頂点iから出ていく遷移を「配る遷移形式」と呼ぶ

図5.8

- 各遷移形式に基づいてDPを構成することを、それぞれ貰うDP、
  配るDPなどと呼んだりする

配るDPでFrog問題を解く

```cpp
int main() {
    // 入力
    int N; cin >> N;
    vector<long long> h(N);
    for (int i = 0; i < N; ++i) cin >> h[i];

    // 初期化（最小化問題なので INF に初期化）
    vector<long long> dp(N, INF);

    // 初期条件
    dp[0] = 0;

    // ループ
    for (int i = 0; i < N; ++i) {
        if (i + 1 < N) {
            chmin(dp[i + 1], dp[i] + abs(h[i] - h[i + 1]));
        }
        if (i + 2 < N) {
            chmin(dp[i + 2], dp[i] + abs(h[i] - h[i + 2]));
        }
    }

    // 答え
    cout << dp[N - 1] << endl;
}
```

- 計算量は貰うDPと同じく$O(N)$
  最初のうちはどちらで解くかは気にしなくて良い

Frog問題を全探索で解く

Code 5.5

- このプログラムは、フィボナッチ数列を再帰で解いたコードと似ている

(再掲)フィボナッチ数列を再帰で解く

- 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

Frog問題を全探索で解く

図5.9

- 再帰関数において明らかに無駄があり、指数時間アルゴリズムとなる
- しかしfiboはメモ化再帰で計算時間を節約できるのであった。
  ならば今回の問題も上手くいく？

Frog問題をメモ化再帰で解く

```cpp
long long rec(int i) {
    // DP の値が更新されていたらそのままリターン
    if (dp[i] < INF) return dp[i];

    // ベースケース: 足場 0 のコストは 0
    if (i == 0) return 0;

    // 答えを表す変数を INF で初期化する
    long long res = INF;

    // 足場 i - 1 から来た場合
    chmin(res, rec(i - 1) + abs(h[i] - h[i - 1]));

    // 足場 i - 2 から来た場合
    if (i > 1) {
        chmin(res, rec(i - 2) + abs(h[i] - h[i - 2]));
    }

    // 結果をメモしながら、返す
    return dp[i] = res;
}

int main() {
    // 入力受け取り
    cin >> N;
    h.resize(N);
    for (int i = 0; i < N; ++i) cin >> h[i];

    // 初期化（最小化問題なので INF に初期化）
    dp.assign(N, INF);

    // 答え
    cout << rec(N - 1) << endl;
}
```

動的計画法の例(1):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

動的計画法の例(1):ナップサック問題

- 全探索を行うとすると、各品物を含むか含まないかを全通り考えるので$O(2^N)$となり、指数時間アルゴリズムとなる。

- これを動的計画法にすることで、計算量の効率化を図りたい。

動的計画法の基本的な考え方:
問題を部分問題に分割する。すなわち、N個の対象物ではなく最初のi個の対象物に対して問題を解く方法を考える。

動的計画法の例(1):ナップサック問題

- まず、次のように問題を切り出してみる

> $\mathrm{dp}[i] \leftarrow$ 最初の i 個の品物 $\{0, 1, \ldots, i-1\}$ までの中から重さが W を超えないように選んだときの，価値の総和の最大値

- しかしこの方法は、
  dp[i+1]とdp[i]の関係を定式化できない為に失敗
  (i+1番目を選ぶことで重さがWを超えるかどうかがわからない)

- 問題が与えられた時、単に動的計画法を使えば良いということではなくどのように定式化するかが重要となる。
  ある程度はパターンがあるので、まずはパターンを把握することが大事
  (例: Frog問題とFib問題の類似性)

動的計画法の例(1):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。

- dp[i]だけでは遷移を考えられなかったので、より粒度を細かくすることで遷移を考えられるようにする
(今回は１次元配列から２次元配列へと拡張している)

- まず初期条件として、全てのwに対してdp[0][w] ＝ 0とする。
(frog問題と違い最小化ではなく最大化するものを求めるため)

- 全てのwに対してdp[i][w]が求まっているとした時、dp[i+1][w]を求めることを考える。

動的計画法の例(1):ナップサック問題

- chmax関数を用いて緩和していくことを考える。

i 番目の品物を選ぶとき：

選んだ後に $(i+1, w)$ の状態になるならば，選ぶ前は $(i, w - \mathrm{weight}[i])$ の状態であり，その状態に価値 $\mathrm{value}[i]$ が加わるので

```
chmax(dp[i+1][w], dp[i][w - weight[i]] + value[i])
```

となります (ただし `w - weight[i] >= 0` の場合のみ).

i 番目の品物を選ばないとき：

選ばないならば，重さも価値も特に変化しないので，

```
chmax(dp[i+1][w], dp[i][w])
```

となります.

動的計画法の例(1):ナップサック問題

- 例: 6個の品物があり、W= 7、
  (weight, value) ={(2, 3), (1, 2), (3, 6), (2, 1), (1, 3), (5, 85)}とする。
- 初期状態として、全てのwに対してdp[0][w] = 0とする(確定)。
  また、他の全ての欄もあらかじめdp[i][w] = 0とされているものとする
  (緩和のための初期化)

図5.10

動的計画法の例(1):ナップサック問題

- dp[1][0]を考える。
  w-weight[i] = 0-2 <0のため、1番目の品物を選ぶ場合は計算されない。

  よって品物を選ばなかった際の
  chmax(dp[1][0], dp[0][0])のみが計算され、これは0。
  同様に、dp[1][1]も0となる。

図5.10

動的計画法の例(1):ナップサック問題

- dp[1][2]を考える。
  w-weight[1] = 2-2 =0のため、1番目の品物を選ぶ場合が計算される。

  chmax(dp[1][2], dp[0][0]+value[0]) = 3
  品物を選ばなかった場合は
  chmax(dp[1][2], dp[0][2]) = 3
  以下d[1][w]は同様。

図5.10

動的計画法の例(1):ナップサック問題

- dp[2][0]は明らかに0、dp[2][1]はweight[2]=1より明らかに2である。

  dp[2][2]は
  品物を選ぶ場合
  chmax(dp[2][2], dp[1][2-1]+value[2]) = chmax(0, 2) = 2
  品物を選ばない場合
  chmax(dp[2][2], dp[1][2]) = chmax(2, 3) = 3

図5.10

動的計画法の例(1):ナップサック問題

- dp[2][3]は
  品物を選ぶ場合
  chmax(dp[2][3], dp[1][3-1]+value[2]) = chmax(0, 5) = 5
  品物を選ばない場合
  chmax(dp[2][3], dp[1][3]) = chmax(5, 3) = 5
  以下dp[2][w]は同様

図5.10

動的計画法の例(1):ナップサック問題

- このような計算を繰り返すことで、表を全て埋めていく

1つのマスを埋める処理はO(1)でできるため、
全体の計算量はO(NW)となる

図5.10

動的計画法の例(1):ナップサック問題

```
19      // DP テーブル定義
20      vector<vector<long long>> dp(N + 1, vector<long long>(W + 1, 0));
21
22      // DPループ
23      for (int i = 0; i < N; ++i) {
24          for (int w = 0; w <= W; ++w) {
25
26              // i 番目の品物を選ぶ場合
27              if (w - weight[i] >= 0) {
28                  chmax(dp[i + 1][w], dp[i][w - weight[i]] + value[i]);
29              }
30
31              // i 番目の品物を選ばない場合
32              chmax(dp[i + 1][w], dp[i][w]);
33          }
34      }
35
36      // 最適値の出力
37      cout << dp[N][W] << endl;
```

文字列間の距離を計算する

- 2つの文字列がどれくらい似ているのかを知りたいとする
  (Webにおけるテキスト検索や、バイオインフォマティクスで必須)

- ハミング距離→長さが同じ文字列に対して、異なっている文字の数

  HANABIとHAWAIIでは、

```
HANABI
||X|X|
HAWAII
```

  のため、ハミング距離は2

文字列間の距離を計算する

- ではHANABIとHNABIAではどうか？

  ```
  HANABI
  |XXXXX
  HNABIA
  ```

  のため、ハミング距離は5となり、結構遠い。

- しかし、文字列だけ見ると非常に類似しているように見える。
  この類似性は、ハミング距離では捉えきれないものなのかもしれない。

- ここで、新たな距離として、文字の削除と挿入を考えた
  編集距離を考える。

文字列間の距離を計算する

編集距離の定義

- 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

編集距離を求める動的計画法:
dp[i][j]に、S[0:i-1]とTの[0:i-1]の編集距離を格納する

- 初期条件はdp[0][0] = 0
  そして、変更・削除・挿入の3つの場合を場合分けして考える。

変更操作 (S の i 文字目と T の j 文字目とを対応させる):

$S[i-1] = T[j-1]$ のとき: コストを増やさずに済みますので
`chmin(dp[i][j], dp[i-1][j-1])` です.

$S[i-1] \neq T[j-1]$ のとき: 変更操作が必要ですので
`chmin(dp[i][j], dp[i-1][j-1] + 1)` です.

動的計画法の例(2):編集距離

削除操作 (S の i 文字目を削除)：

S の i 文字目を削除する操作を行いますので
`chmin(dp[i][j], dp[i-1][j] + 1)` です.

挿入操作 (T の j 文字目を削除)：

T の j 文字目を削除する操作を行いますので
`chmin(dp[i][j], dp[i][j-1] + 1)` です.

- この手続きの元、”logistic”と”algorithm”という2つの文字列を比較した場合の遷移は次のようなグラフで表現できる。
  なお、計算量はO(|S||T|)

動的計画法の例(2):編集距離

図5.12

動的計画法の例(2):編集距離

```
19      // DP テーブル定義
20      vector<vector<int>> dp(S.size() + 1, vector<int>(T.size() + 1, INF));
21
22      // DP 初期条件
23      dp[0][0] = 0;
24
25      // DPループ
26      for (int i = 0; i <= S.size(); ++i) {
27          for (int j = 0; j <= T.size(); ++j) {
28              // 変更操作
29              if (i > 0 && j > 0) {
30                  if (S[i - 1] == T[j - 1]) {
31                      chmin(dp[i][j], dp[i - 1][j - 1]);
32                  }
33                  else {
34                      chmin(dp[i][j], dp[i - 1][j - 1] + 1);
35                  }
36              }
37
38              // 削除操作
39              if (i > 0) chmin(dp[i][j], dp[i - 1][j] + 1);
40
41              // 挿入操作
42              if (j > 0) chmin(dp[i][j], dp[i][j - 1] + 1);
43          }
44      }
```

動的計画法の例(3):区間分割最適化

- 1列に並んだ対象物をある基準で区間に分割する

図5.14

- 日本語の分かち書きなどの応用例がある
  →
  「日本語／の／分かち書き／など／の／応用例／が／ある」

動的計画法の例(3):区間分割最適化

- なお、区間は次のように表されるものとする。

図5.15

動的計画法の例(3):区間分割最適化

- 区間分割最適化問題を、次のように定義する。

区間分割の仕方を最適化する問題

動的計画法の例(3):区間分割最適化

- 区間分割最適化問題も動的計画法で解ける

> 区間分割の動的計画法:
> dp[i]に区間[0, i)の最小分割コストを格納する。

- 初期条件はdp[0] = 0
- dp[0]からdp[i-1]までが計算されていたとする。
  区間iまでの分割は、区間jまでの分割+$c_{j,i}$であることを考えると、
  緩和式は次の通り

```
chmin(dp[i], dp[j] + c[j][i])
```

ただしj < i

動的計画法の例(3):区間分割最適化

```
24        // DP テーブル定義
25        vector<long long> dp(N + 1, INF);
26
27        // DP 初期条件
28        dp[0] = 0;
29
30        // DPループ
31        for (int i = 0; i <= N; ++i) {
32            for (int j = 0; j < i; ++j) {
33                chmin(dp[i], dp[j] + c[j][i]);
34            }
35        }
```

- 計算量は$O(N^2)$となる。

まとめ

- 問題を解くための設計技法として、動的計画法を紹介した。

- 動的計画法とは、与えられた問題を部分問題に分解し、
  部分問題の解をメモ化しながら、
  その解を統合することで元の与えられた問題の解を解く方法である。

- 緩和、貰う遷移形式、配る遷移形式の概念を紹介した。

- 動的計画法の例題として、Frog問題、ナップザック問題、
  編集距離を計算する問題、区間分割問題などを紹介した。