第4章

動的計画法

4.1 問題 B16：Frog 1 (難易度：★2相当)

この問題は、いきなり足場 1 から足場 N までの最小コストを求めると難しくなってしまいます。しかし、$i = 1,2,\ldots,N$ の順に「足場 1 から足場 i までの最小コストはいくつか？」ということを考えていくと、計算量 $O(N)$ で解くことができます。

管理する配列

`dp[i]`：足場 1 から足場 `i` まで移動するための最小コスト

配列 dp の計算

まず、明らかに $\mathrm{dp}[1] = 0$ および $\mathrm{dp}[2] = |H_1 - H_2|$ が成り立ちます。次に $\mathrm{dp}[3]$ 以降ですが、**最後の行動で場合分け**をすると、足場 i に移動する方法としては以下の 2 つが考えられます：

- **方法A：** 1 回で、足場 $i-1$ から足場 i に移動する
- **方法B：** 1 回で、足場 $i-2$ から足場 i に移動する

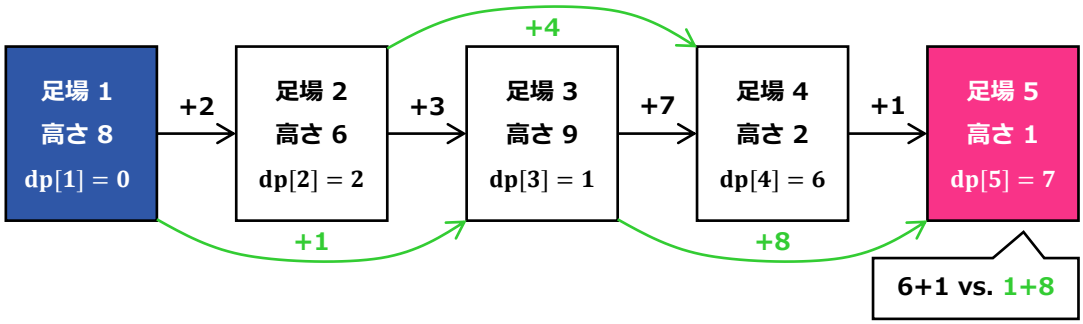
ここで、方法 A をとったときの合計コストは $\mathrm{dp}[i-1] + |H_{i-1} - H_i|$、方法 B をとったときの合計コストは $\mathrm{dp}[i-2] + |H_{i-2} - H_i|$ であるため、$\mathrm{dp}[3]$ 以降は次の式によって計算することができます。

```
dp[i] = min(dp[i-1] + abs(H[i-1]-H[i]),
            dp[i-2] + abs(H[i-2]-H[i]))
```

（次ページへ続きます）

具体例

たとえば、$N = 5, A = [8, 6, 9, 2, 1]$ の場合、次のような計算により、答えが $\mathrm{dp}[5] = 7$ であると分かります。なお、矢印に書かれた整数は、1 回の移動コスト（$|H_{i-1} - H_i|$ または $|H_{i-2} - H_i|$）となっています。



解答例（C++）

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

int N, H[100009];
int dp[100009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> H[i];

    // 動的計画法
    dp[1] = 0;
    dp[2] = abs(H[1] - H[2]);
    for (int i = 3; i <= N; i++) {
        dp[i] = min(dp[i-1] + abs(H[i-1] - H[i]), dp[i-2] + abs(H[i-2] - H[i]));
    }

    // 出力
    cout << dp[N] << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

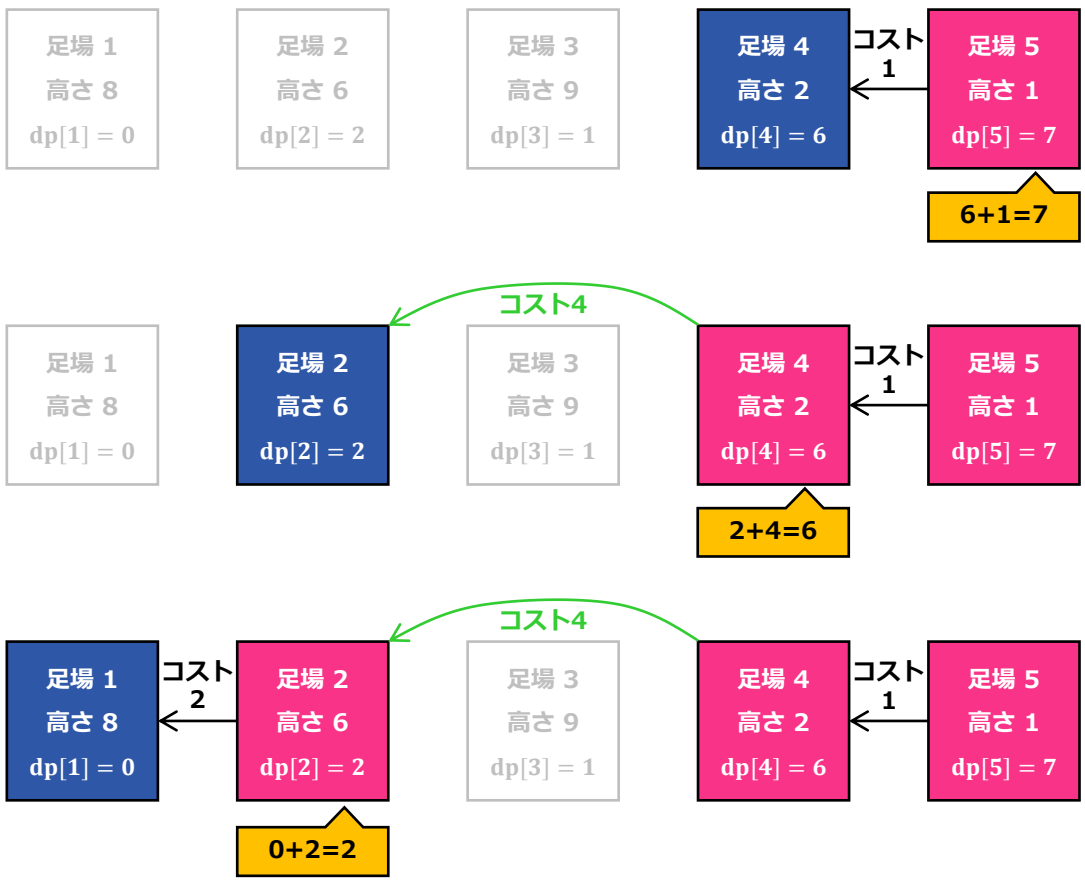
4.2 問題 B17：Frog 1 with Restoration (難易度：★3相当)

$\mathrm{dp}[i]$ の値を計算した後は、問題 A17 と同様に**ゴールから考えていくこと**で、「具体的な経路」を得ることができます。具体的には、いま部屋 i にいるとき以下のようになります。

- $\mathrm{dp}[i] = \mathrm{dp}[i-1] + |h_{i-1} - h_i|$ の場合：部屋 $i-1$ に進むのが最適
- $\mathrm{dp}[i] = \mathrm{dp}[i-2] + |h_{i-2} - h_i|$ の場合：部屋 $i-2$ に進むのが最適

具体例

たとえば、$N = 5, h = [8, 6, 9, 2, 1]$ の場合、次のような計算により、1→2→4→5 という経路が最適だと分かります。



解答例（C++）

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

int N, H[100009];
int dp[100009];
vector<int> Answer;

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> H[i];

    // 動的計画法
    dp[1] = 0;
    dp[2] = abs(H[1] - H[2]);
    for (int i = 3; i <= N; i++) {
        dp[i] = min(dp[i-1] + abs(H[i-1] - H[i]), dp[i-2] + abs(H[i-2] - H[i]));
    }

    // 動的計画法の復元
    int Place = N;
    while (true) {
        Answer.push_back(Place);
        if (Place == 1) break;

        // どちらに移動するかを求める
        if (dp[Place-1] + abs(H[Place-1] - H[Place]) == dp[Place]) Place = Place-1;
        else Place = Place - 2;
    }
    reverse(Answer.begin(), Answer.end());

    // 答えを求める
    cout << Answer.size() << endl;
    for (int i = 0; i < Answer.size(); i++) {
        if (i) cout << " ";
        cout << Answer[i];
    }
    cout << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

4.3 問題 B18：Subset Sum 2 (難易度：★4相当)

応用問題 4.2 と同様、「カード N を選ぶべきか？」「カード $N-1$ を選ぶべきか？」「カード $N-2$ を選ぶべきか？」という順番で答えを求めていくと、 合計を S にするカードの選び方が分かります。

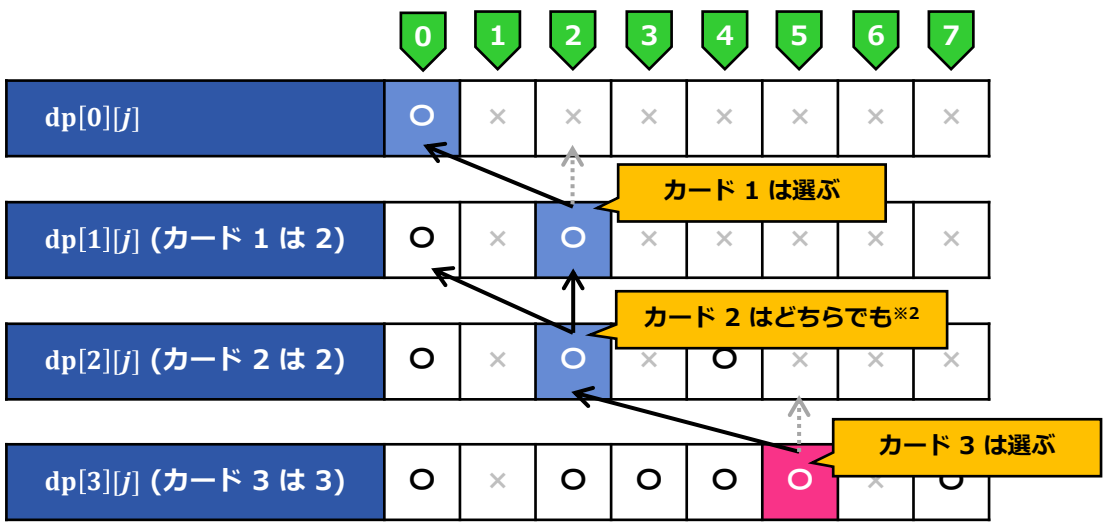
たとえばカード N を選ぶかどうかは、以下のように決めれば良いです。カード $N-1$ 以降についても同様です。

- **$\mathrm{dp}[N][S]$ がマルの場合：**カード N を選ばない※1
- **$\mathrm{dp}[N][S-A_N]$ がマルの場合：**カード N を選ぶ

※2 つのうち、どちらか片方の条件は必ず満たす

具体例

たとえば、$N=5, S=5, A=[2,2,3]$ の場合、次のようにして「カード 1 と 3 を選べば良い」ということが分かります。



※1　突然 $\mathrm{dp}[N][S]$ などが出てきて混乱するかもしれませんが、動的計画法の復元を行う前に、本の 116 ページの通りに配列 dp の値を計算しなければならないことに注意してください

※2　どちらでもと書いてありますが、ここでは「カード 2 を選ばない」方を選択していることに注意

解答例（C++）

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int N, S, A[69];
bool dp[69][10009];
vector<int> Answer;

int main() {
    // 入力
    cin >> N >> S;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 動的計画法 (i = 0)
    dp[0][0] = true;
    for (int i = 1; i <= S; i++) dp[0][i] = false;
    // 動的計画法 (i >= 1)
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j <= S; j++) {
                if (j < A[i]) {
                        if (dp[i - 1][j] == true) dp[i][j] = true;
                        else dp[i][j] = false;
                }
                if (j >= A[i]) {
                        if (dp[i-1][j]==true || dp[i-1][j-A[i]]==true) dp[i][j]=true;
                        else dp[i][j] = false;
                }
        }
    }

    // 選び方が存在しない場合
    if (dp[N][S] == false) { cout << "-1" << endl; return 0; }

    // 答えの復元（Place は "現在の総和"）
    int Place = S;
    for (int i = N; i >= 1; i--) {
        if (dp[i - 1][Place] == true) Place = Place - 0; // カード i を選ばない
        else {
                Place = Place - A[i]; // カード i を選ぶ
                Answer.push_back(i);
        }
    }
    reverse(Answer.begin(), Answer.end());

    // 出力
    cout << Answer.size() << endl;
    for (int i = 0; i < Answer.size(); i++) {
        if (i >= 1) cout << " ";
        cout << Answer[i];
    }
    cout << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

4.4 問題 B19：Knapsack 2 (難易度：★4相当)

例題で扱ったナップザック問題では、W の値が小さいため、「どの品物まで決めたか」「現在の合計重量」の 2 つを持つ動的計画法を考えました。

一方、今回は価値 v_i の値が小さいので、**「どの品物まで決めたか」「現在の合計価値」の 2 つ**を持つと、上手くいきます。

管理する配列

dp[i][j]：品物 1,2,…,i の中から、価値の合計が j となるように選ぶことを考える。このとき、合計重量としてあり得る最小値はいくつか。

※ v_i の合計は高々 100000 なので、$j \leq 100000$ まで考えれば良い

答えとなる値

$\mathrm{dp}[N][i] \leq W$ を満たす最大の i が、求める答え（合計価値の最大値）となります。

配列 dp の計算

まず、明らかに $\mathrm{dp}[0][0] = 0$ となります。$\mathrm{dp}[0][1], \mathrm{dp}[0][2], \ldots$ の値は、そもそも選び方が存在しないので ∞ としておきます。

次に、$i \geq 1$ に対する $\mathrm{dp}[i][j]$ の値はどうやって計算すれば良いのでしょうか。最後の行動で場合分けすると、$\mathrm{dp}[i][j]$ の状態になる方法として以下の 2 つがあると分かります。

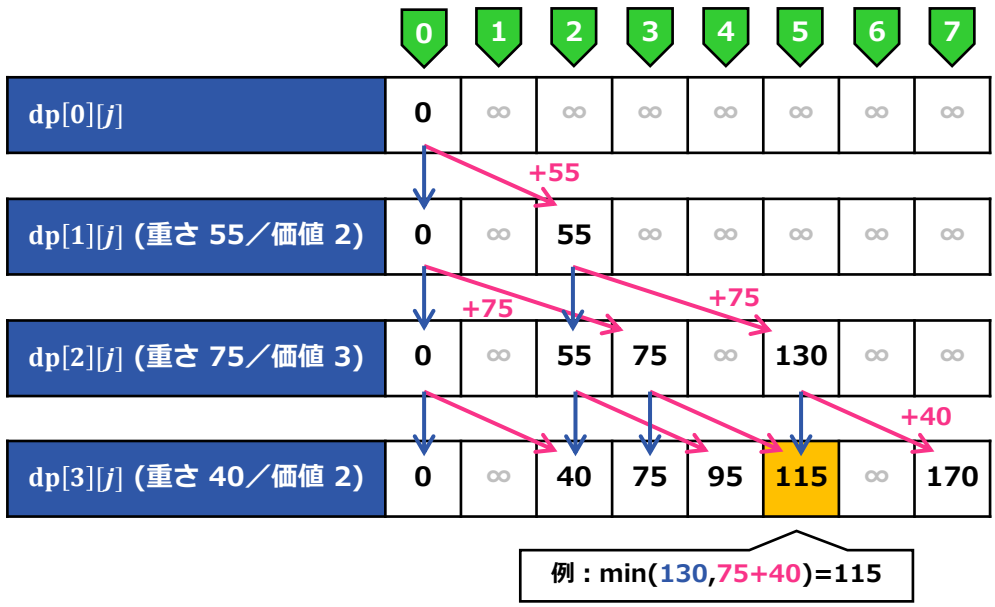
選び方	合計重量の最小値
方法A：品物 $i-1$ 時点で合計価値 j／品物 i を買わない	$\mathrm{dp}[i-1][j]$
方法B：品物 $i-1$ 時点で合計価値 $j-v_i$／品物 i を買う	$\mathrm{dp}[i-1][j-v_i]+w_i$

したがって、$\mathrm{dp}[i][j]$ の値は次のようにして計算することができます。

```
dp[i][j] = min(dp[i-1][j], dp[i-1][j-v[i]]+w[i])
```

具体例

たとえば、$N = 5, W = 100, (w_i, v_i) = (55,2), (75,3), (40,2)$ の場合、次のようにして配列 dp の値を計算することができます。$\mathrm{dp}[5][4] \leq 100$ なので、問題の答え（価値の最大値）は 4 です。



解答例（C++）

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

long long N, W, w[109], v[109];
long long dp[109][100009];

int main() {
    // 入力・配列の初期化
    cin >> N >> W;
    for (int i = 1; i <= N; i++) cin >> w[i] >> v[i];
    for (int i = 0; i <= N; i++) {
        for (int j = 0; j <= 100000; j++) dp[i][j] = 1'000'000'000'000'000LL;
    }
```

```
    // 動的計画法
    dp[0][0] = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j <= 100000; j++) {
                if (j < v[i]) dp[i][j] = dp[i - 1][j];
                else dp[i][j] = min(dp[i - 1][j], dp[i - 1][j - v[i]] + w[i]);
        }
    }

    // 答えの出力
    long long Answer = 0;
    for (int i = 0; i <= 100000; i++) {
        if (dp[N][i] <= W) Answer = i;
    }
    cout << Answer << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

4.5 問題 B20：Edit Distance (難易度：★6相当)

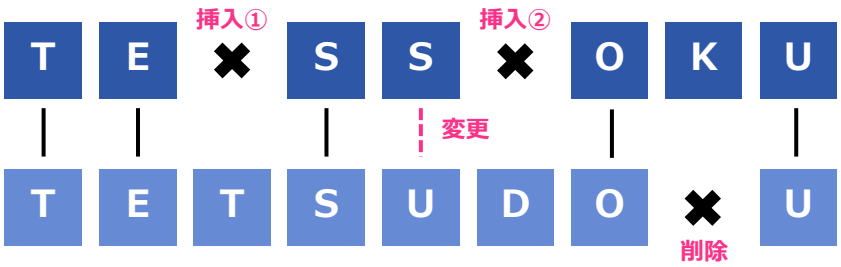
まず、文字列 S を編集する操作は、**文字列 S,T を一列に並べる操作**に対応します。具体的には以下のようになります。

操作1：文字列 S から 1 つの文字を削除	**S の下に T の文字を置かない**
操作2：文字列 S の 1 つの文字を変更	**S と T の同じ位置に異なる文字を置く**
操作3：文字列 S に 1 つの文字を挿入	**T の上に S の文字を置かない**

具体例

たとえば、文字列 S = “tessoku” を T = “tetsudou” に変更する操作の一例※3は、以下のような文字列の並びに対応します。

操作回数は、文字を置かなかった個数と、文字が一致していない個数を合計して 4 回です。



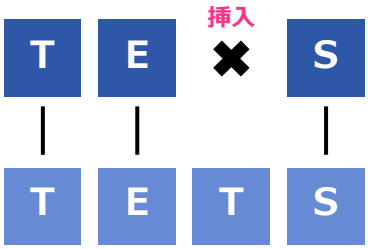
動的計画法へ

そこで、操作回数の最小値、すなわち**「文字を置かなかった個数と、文字が一致していない個数の合計」**（以下、合計コストとする）の最小値はどうやって計算すれば良いのでしょうか。まずは以下のような動的計画法を考えます。

`dp[i][j]`：文字列 S の `i` 文字目、文字列 T の `j` 文字目まで並べたとき、その時点での合計コストの最小値はいくつか。※4

※3 “tessoku” → “tetssoku”（挿入①）→ “tetssdoku”（挿入②）→ “tetsudoku”（変更）→”tetsudou”（削除）という操作の場合
※4 文字列は左から順番に並べることを仮定している

たとえば S = “tessoku”、T = “tetsudou” の場合、$\mathrm{dp}[3][4] = 1$ となります。なぜなら、S の 3 文字目まで（tes まで）と T の 4 文字目まで（tets まで）は、以下のようにしてコスト 1 で並べることができるからです。



配列 dp の計算

それでは、配列 dp の値を計算することを考えましょう。まず明らかに dp[0][0]=0 です。また dp[i][j] が指す状態に遷移する方法としては以下の 4 つが考えられます。

操作	合計コストの最小値
削除操作（S の下に T を置かない）	dp[i-1][j]+1
挿入操作（T の下に S を置かない）	dp[i][j-1]+1
変更操作（S,T で異なる文字を置く）	dp[i-1][j-1]+1　※ $S_i \neq T_j$ の場合
何も変えない（S,T で同じ文字を置く）	dp[i-1][j-1]　※ $S_i = T_j$ の場合

したがって、配列 dp の値は次のように計算すれば良いです。

$S_i = T_j$ の場合：

- dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1])

$S_i \neq T_j$ の場合：

- dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1)

以上の内容を実装すると、次ページの解答例のようになります。ここで求める答えは、S,T の長さをそれぞれ N,M とするとき、dp[N][M] であることに注意してください。

解答例（C++）

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

int N, M, dp[2009][2009];
string S, T;

int main() {
    // 入力
    cin >> S; N = S.size();
    cin >> T; M = T.size();

    // 動的計画法
    dp[0][0] = 0;
    for (int i = 0; i <= N; i++) {
        for (int j = 0; j <= M; j++) {
                if (i >= 1 && j >= 1 && S[i - 1] == T[j - 1]) {
                        dp[i][j] = min({dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]});
                }
                else if (i >= 1 && j >= 1) {
                        dp[i][j] = min({dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1});
                }
                else if (i >= 1) {
                        dp[i][j] = dp[i - 1][j] + 1;
                }
                else if (j >= 1) {
                        dp[i][j] = dp[i][j - 1] + 1;
                }
        }
    }

    // 出力
    cout << dp[N][M] << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

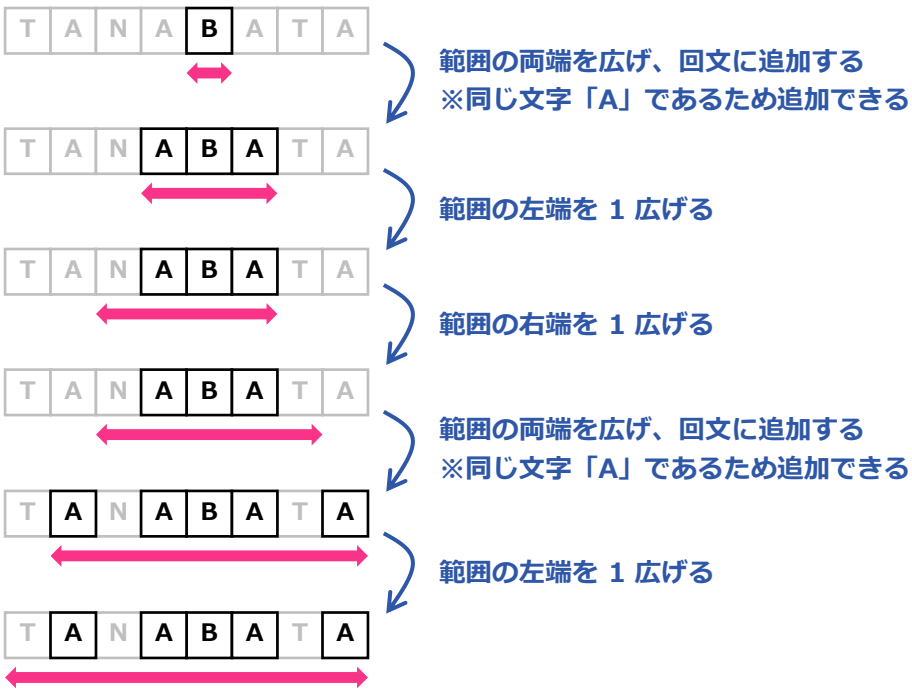
4.6 問題 B21：Longest Subpalindrome (難易度：★6相当)

まず、文字列の中から回文を取り出す操作は、以下の操作に対応します。

- 最初に 1 つまたは 2 つの文字を選び、回文に追加する（2 つの文字を選ぶ場合は、同じ文字でなければならない）。
- その後、少しずつ範囲を広げていき、**「範囲の両端の文字が同じであれば、それらの文字を回文に追加する」**という操作を繰り返す。

たとえば文字列“tanabata”から回文“aabaa”を取り出す操作は、以下のような操作手順に対応します。

なお、この図では、回文として追加された部分を黒で表示しています。また、範囲を赤い矢印で示しています。



ここで、最初に 2 文字を選ぶ場合は、連続する 2 文字であると仮定してもかまいません。なぜなら、連続しない 2 文字を選ぶのは最適ではないからです（間の 1 文字を選ぶと、回文の長さが 1 増えます）。※5

※5　たとえば文字列 “kazan” の 2 文字目・4 文字目を最初に選ぶのは最適ではありません。なぜなら、3 文字目の「k」を追加すると、回文の長さが 1 増えるからです。

動的計画法を考える

そこで、最も多くの文字を回文として追加するにはどうすれば良いのでしょうか。以下のような動的計画法を考えます。

`dp[l][r]`：文字列の `l` 文字目から `r` 文字目までが範囲になっているとき、既に最大何文字を回文として追加できているか。

配列 dp の計算

まず、初期状態は以下のようになります。

- 1 文字を選ぶ場合：`dp[i][i]=1`
- 2 文字を選ぶ場合：`dp[i][i+1]=2`（$S_i = S_{i+1}$ の場合）

次に、状態遷移を考えます。`dp[l][r]` の状態に遷移する方法として考えられるものは、以下の 3 つです。

操作	累計文字数の最大値
左端を 1 広げる	`dp[l+1][r]`
右端を 1 広げる	`dp[l][r-1]`
左端・右端を 1 広げ、回文に追加する	`dp[l+1][r-1]+2`　　※ $S_l = S_r$ の場合

したがって、`dp[l][r]` の値は以下のようになります。

$S_l = T_t$ の場合：

- `dp[l][r] = max(dp[l][r-1], dp[l+1][r], dp[l+1][r-1]+2)`

$S_l \neq T_t$ の場合：

- `dp[l][r] = max(dp[l][r-1], dp[l+1][r])`

ここまでの内容を実装すると、次ページの解答例のようになります。計算量は $O(N^2)$ です。なお、解答例では `r-1` の小さい順に `dp[l][r]` の値を計算していることに注意してください。

解答例（C++）

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

int N;
int dp[1009][1009];
string S;

int main() {
    // 入力
    cin >> N;
    cin >> S;

    // 動的計画法（初期状態）
    for (int i = 0; i < N; i++) dp[i][i] = 1;
    for (int i = 0; i < N - 1; i++) {
        if (S[i] == S[i + 1]) dp[i][i + 1] = 2;
        else dp[i][i + 1] = 1;
    }

    // 動的計画法（状態遷移）
    for (int LEN = 2; LEN <= N - 1; LEN++) {
        for (int l = 0; l < N - LEN; l++) {
                int r = l + LEN;

                if (S[l] == S[r]) {
                        dp[l][r] = max({ dp[l][r-1], dp[l+1][r], dp[l+1][r-1]+2 });
                }
                else {
                        dp[l][r] = max({dp[l][r-1], dp[l+1][r] });
                }
        }
    }

    // 答えを求める
    cout << dp[0][N - 1] << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

4.7 問題 B22：Frog 1 with Restoration (難易度：★3相当)

足場 i にいるときの一手先の行動としては、「足場 $i+1$ に移動する」「足場 $i+2$ に移動する」の 2 つが考えられます。

そのため、足場 1 から足場 i まで移動するための最小コストを `dp[i]` とするとき、`dp[i]` の値は以下のようにして計算することができます（問題 A16 とは異なり、配る遷移形式を使っています）。

最初、$\mathrm{dp}[1] = 0$ に設定し、$\mathrm{dp}[2], \mathrm{dp}[3], \ldots, \mathrm{dp}[N] = \infty$ とする。
その後、$i = 1,2, \ldots, N$ の順に、以下の操作を行う。

- $\mathrm{dp}[i+1]$ を $\min(\mathrm{dp}[i+1], \mathrm{dp}[i] + |h_i - h_{i+1}|)$ に更新する
- $\mathrm{dp}[i+2]$ を $\min(\mathrm{dp}[i+2], \mathrm{dp}[i] + |h_i - h_{i+2}|)$ に更新する

解答例（C++）

```cpp
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;
int N, H[100009], dp[100009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> H[i];

    // 配列 dp の初期化
    dp[1] = 0;
    for (int i = 2; i <= N; i++) dp[i] = 2000000000;

    // 動的計画法
    for (int i = 1; i <= N; i++) {
        if (i<=N-1) dp[i+1] = min(dp[i+1], dp[i]+abs(H[i+1]-H[i])); // 足場 i+1 に行く
        if (i<=N-2) dp[i+2] = min(dp[i+2], dp[i]+abs(H[i+2]-H[i])); // 足場 i+2 に行く
    }
    cout << dp[N] << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

4.8 問題 B23：Traveling Salesman (難易度：★5相当)

まず考えられる方法は、移動方法 $N!$ 通りを全探索することです。しかし本問題の制約は $N \leq 15$ であり、$15!$ は 10^{12} を超えるため、残念ながら実行時間制限に間に合いません。

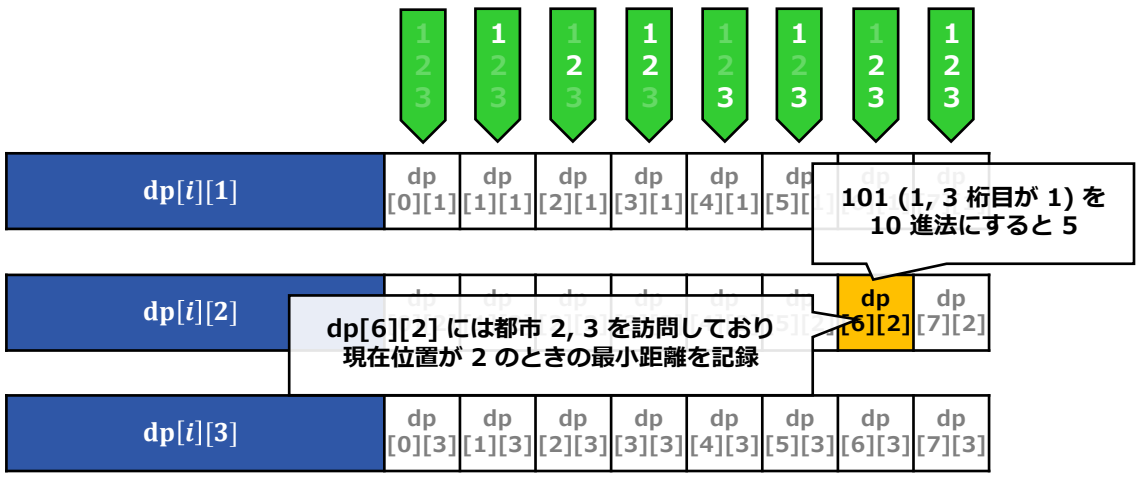
動的計画法（ビットDP）を考える

そこで、次のような動的計画法を考えます。既に訪問した都市 `i` は、整数ではなく集合であることに注意してください。

> `dp[i][j]`：既に訪問した都市の集合が `i` であり、現在位置が `j` であるときの、現時点での最小移動距離

なお、プログラム上では配列の添字として集合を設定することができず、整数にする必要があります。

整数にする方法としては、本の 141 ページ（4.8 節）に記したように、2 進法を使うなどの方法があります（下図参照）。



配列 dp はどうやって計算するか

まず、初期状態は dp[1][1]=0 です※6。なぜなら、**最初は都市 1 から出発すると考えても一般性を失わない**からです。

次に、状態遷移はどのようにして行えば良いのでしょうか。配る遷移形式にしたがって考えると、次に訪れる都市を k とし、都市 j と k の間の距離を dist(j,k) とするとき、以下のようになります。

```
dp[i+(1<<k)][k] = min(dp[i+(1<<k)][k], dp[i][j] + dist(j,k));
```

集合 i + 都市 k（値は $i+2^k$）

ここまでの内容を実装すると、次ページの解答例のようになります。答えは $\mathrm{dp}[2^N-1][1]$ であることに注意してください。

なお、配列 dp の要素数は $N \times 2^N$ であり、各要素につき計算量 $O(N)$ かかるので、プログラム全体の計算量は $O(N^2 \times 2^N)$ です。

解答例（C++）

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

int N, X[19], Y[19];
double dp[1 << 16][19];

int main() {
    // 入力
    cin >> N;
    for (int i = 0; i < N; i++) cin >> X[i] >> Y[i];

    // 配列 dp の初期化
    for (int i = 0; i < (1 << N); i++) {
        for (int j = 0; j < N; j++) dp[i][j] = 1e9;
    }

    // 動的計画法（dp[通った都市][今いる都市] となっている）
    dp[0][0] = 0;
    for (int i = 0; i < (1 << N); i++) {
        for (int j = 0; j < N; j++) {
                if (dp[i][j] >= 1e9) continue;
```

※6　最後に「スタート地点」に戻るため、ここではスタート地点を「訪問した都市の集合」に含めていません

```
                // 都市 j から都市 k に移動したい！
                for (int k = 0; k < N; k++) {
                        // 既に都市 k を通っていた場合
                        if ((i / (1 << k)) % 2 == 1) continue;

                        // 状態遷移
                        double DIST = sqrt(1.0*(X[j]-X[k])*(X[j]-X[k]) + 1.0*(Y[j]-Y[k])*(Y[j]-Y[k]));
                        dp[i+(1<<k)][k] = min(dp[i+(1<<k)][k], dp[i][j] + DIST);
                }
        }
    }

    // 答えを出力
    printf("%.12lf¥n", dp[(1 << N) - 1][0]);
    return 0;
}
```

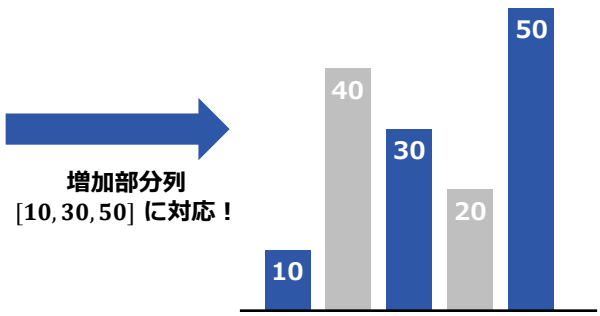
※Python のコードはサポートページをご覧ください

4.9 問題 B24：Many Boxes (難易度：★5相当)

まずは手始めに、$X_1 < X_2 < \cdots < X_N$ を満たす場合を考えます。少し衝撃的かもしれませんが、このようなケースでは、答えは**列 $[Y_1, Y_2, \ldots, Y_N]$ の最長増加部分列問題の答え**と一致します。

なぜなら、1 つの「箱の選び方」が、1 つの「最長増加部分列」に対応するからです※7。箱 1・3・5 を選ぶ場合の具体例を以下に示します（注：最長増加部分列については、本の 144 ページをご覧ください）。

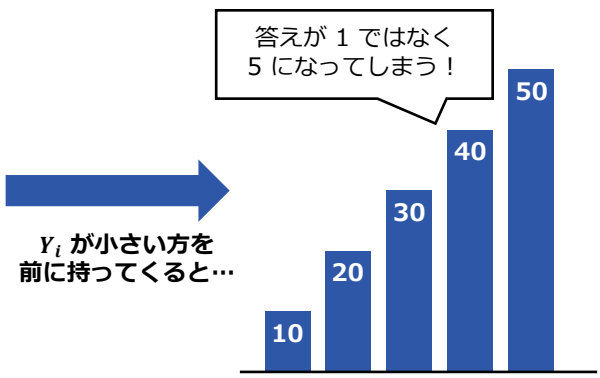
箱の番号	大きさ
箱1	$(X_1, Y_1) = (10, 10)$
箱2	$(X_2, Y_2) = (20, 40)$
箱3	$(X_3, Y_3) = (30, 30)$
箱4	$(X_4, Y_4) = (40, 20)$
箱5	$(X_5, Y_5) = (50, 50)$



それでは、 $X_1 < X_2 < \cdots < X_N$ を満たさない場合はどうでしょうか。箱の番号は答えに関係ないので、あらかじめ X_i の小さい順に箱をソートしておけば良いです。

ただし、縦の長さ X_i が同じである場合は、横の長さ Y_i が大きい方を前に持ってくる必要があることに注意が必要です（下図参照）。

箱の番号	大きさ
箱1	$(X_1, Y_1) = (10, 10)$
箱2	$(X_2, Y_2) = (10, 20)$
箱3	$(X_3, Y_3) = (10, 30)$
箱4	$(X_4, Y_4) = (10, 40)$
箱5	$(X_5, Y_5) = (10, 50)$



※7　必ず増加部分列になる理由は、「箱を箱の中に入れるためには、縦の長さも横の長さも真に短くなっていなければならないこと」から分かります

解答例（C++）

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

int N, X[100009], Y[100009];
int LEN, L[100009];

// 配列 A の最長増加部分列（LIS）の長さを計算する
// 配列 dp を使わない実装方法を利用している
int Get_LISvalue(vector<int> A) {
	LEN = 0;
	for (int i = 1; i <= A.size(); i++) L[i] = 0;

	// 動的計画法
	for (int i = 0; i < A.size(); i++) {
		int pos = lower_bound(L + 1, L + LEN + 1, A[i]) - L;
		L[pos] = A[i];
		if (pos > LEN) LEN += 1;
	}
	return LEN;
}

int main() {
	// 入力
	cin >> N;
	for (int i = 1; i <= N; i++) cin >> X[i] >> Y[i];

	// ソート
	vector<pair<int, int>> tmp;
	for (int i = 1; i <= N; i++) tmp.push_back(make_pair(X[i], -Y[i]));
	sort(tmp.begin(), tmp.end());

	// 求める LIS の配列は？
	vector<int> A;
	for (int i = 0; i < tmp.size(); i++) {
		A.push_back(-tmp[i].second);
	}

	// 出力
	cout << Get_LISvalue(A) << endl;
	return 0;
}
```

※Python のコードはサポートページをご覧ください