第 8 章

データ構造とクエリ処理

8.1 問題 B51：Bracket

(難易度：★4相当)

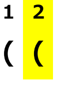
※この問題は例題と比べて相当難しいです。

　まず、カッコ列 (((())(()) について対応関係を列挙することを考えましょう。左から順番に調べていくと、以下のようにして 4 つの対応関係（3-4 文字目・2-5 文字目・6-7 文字目・1-8 文字目）が分かります。

1

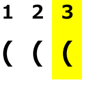
1 文字目は '(' である

2



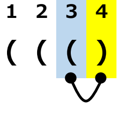
2 文字目は '(' である

3



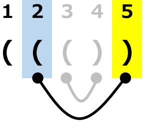
3 文字目は '(' である

4



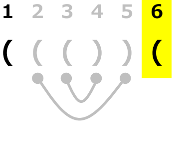
4 文字目は ')' である
残っている一番右の '(' の
3 文字目と対応させる

5



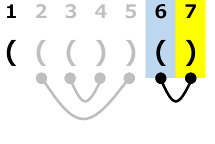
5 文字目は ')' である
残っている一番右の '(' の
2 文字目と対応させる

6



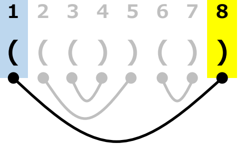
6 文字目は '(' である

7



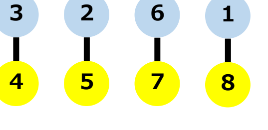
7 文字目は ')' である
残っている一番右の '(' の
6 文字目と対応させる

8



8 文字目は ')' である
残っている一番右の '(' の
1 文字目と対応させる

9



これですべての対応が
わかった！

どう実装するか

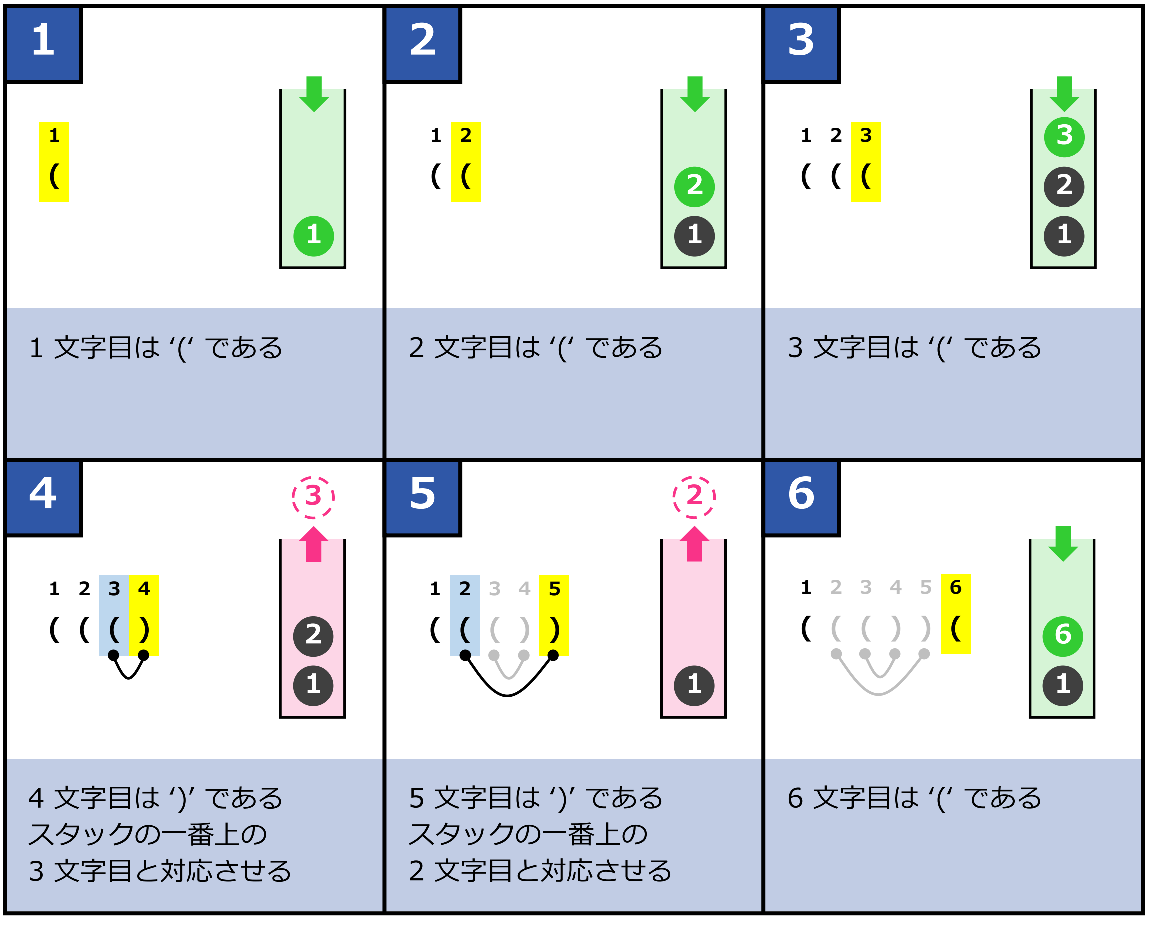
先程の例で説明したように、カッコ列の対応関係は、**「残っている一番右の‘(‘」を調べること**によって列挙できます。

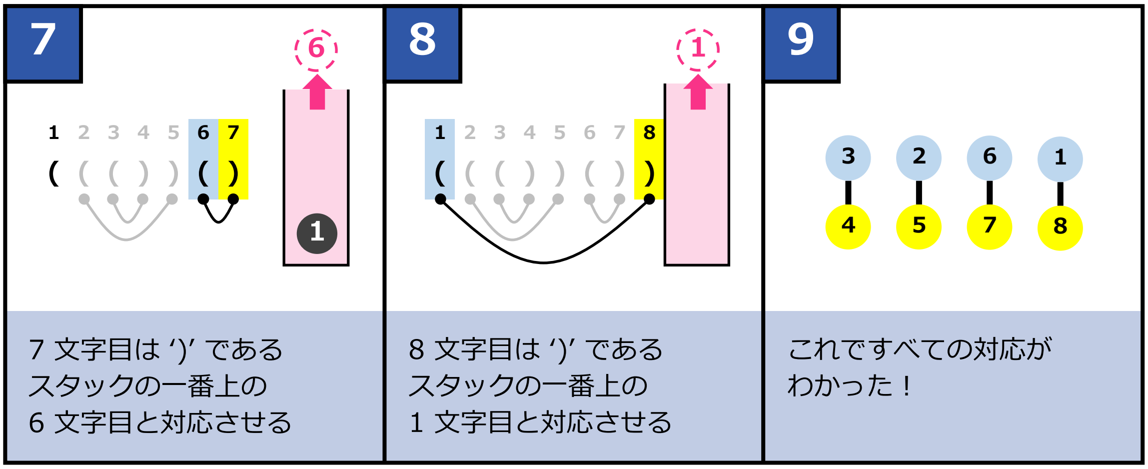
それでは、残っている一番右の ‘(‘ はどうやって効率的に調べられるのでしょうか。もちろん直接調べても良いですが、1 文字当たり計算量 $O(N)$、すなわち全体で計算量 $O(N^2)$ を要し、実行時間制限に間に合いません。

そこで、スタックに**「今残っている ‘(‘ の位置」を左から順に記録する**と効率的です。具体的なアルゴリズムの流れは以下の通りです。

> 左から順番に一文字ずつ読んでいき、次のことを行う。
>
> - **i 文字目が ‘(‘ のとき：** スタックに i を追加する
> - **i 文字目が ‘)’ のとき：** スタックの一番上と i 文字目が対応することが分かる。その後、スタックの一番上の要素を消す。

たとえば、カッコ列が (()))(()) の場合、アルゴリズムの挙動は以下のようになります（最初に説明したものと同じ例です）。





ここまでの内容を実装すると、以下の解答例のようになります。計算量は $O(N)$ です。

解答例（C++）

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

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

    // 左から順番に見ていく
    // 文字列は 0 文字目から始まることに注意
    stack<int> Stack;
    for (int i = 0; i < S.size(); i++) {
        if (S[i] == '(') {
                Stack.push(i + 1);
        }
        if (S[i] == ')') {
                cout << Stack.top() << " " << i + 1 << endl;
                Stack.pop();
        }
    }
    return 0;
}
```

※Python のコードはサポートページをご覧ください

8.2 問題 B52：Ball Simulation （難易度：★3相当）

この問題は、問題文の通りに直接シミュレーションすることで解くことができます。実装例は以下のようになります。計算量は $O(N)$ です。

解答例（C++）

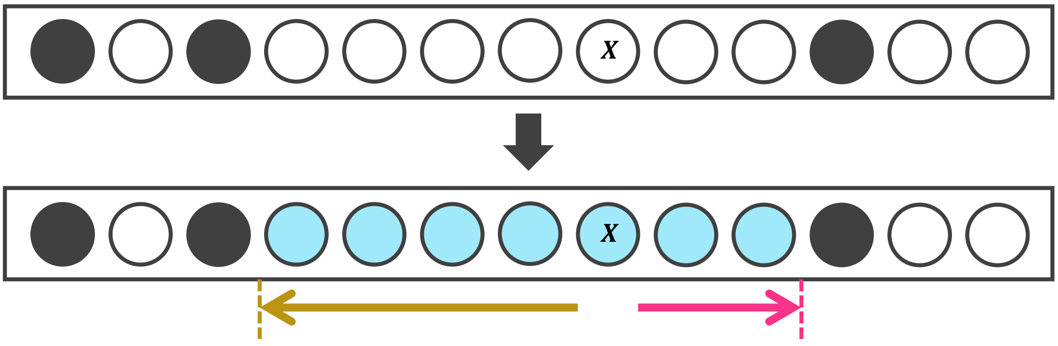
```cpp
#include <iostream>
#include <queue>
using namespace std;

int N, X;
char A[100009];
queue<int> Q;

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

    // シミュレーション
    Q.push(X); A[X] = '@';
    while (!Q.empty()) {
        int pos = Q.front(); Q.pop();
        if (pos - 1 >= 1 && A[pos - 1] == '.') {
                A[pos - 1] = '@';
                Q.push(pos - 1);
        }
        if (pos + 1 <= N && A[pos + 1] == '.') {
                A[pos + 1] = '@';
                Q.push(pos + 1);
        }
    }

    // 出力
    for (int i = 1; i <= N; i++) cout << A[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

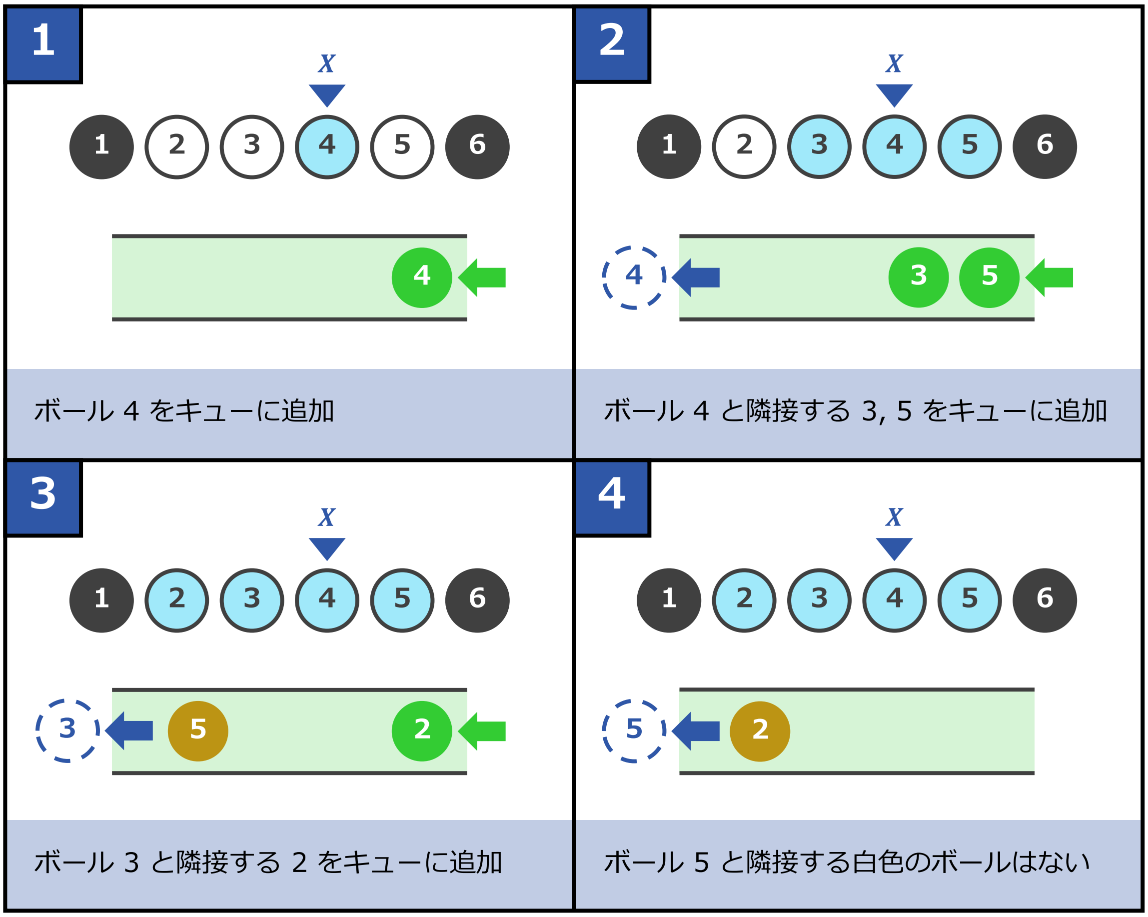
（解説は次ページへ続く）

どの部分が青く塗られるか？

それでは、このシミュレーションではどの部分が青く塗られるのでしょうか。実は、**ボール X から左に進んだときに黒にぶつかるまでの領域**と、**ボール X から右に進んだときに黒にぶつかるまでの領域**だけが、青く塗られます。具体例を以下に示します。



なぜなら、まずボール X がキューに追加され、ボール X と隣り合うボールがキューに追加され、それと隣り合うボールがキューに追加され、・・・ といったように、**黒い “壁” にぶつかるまで連鎖的に「キューに追加される領域」が拡大していく**からです。具体例を以下に示します。

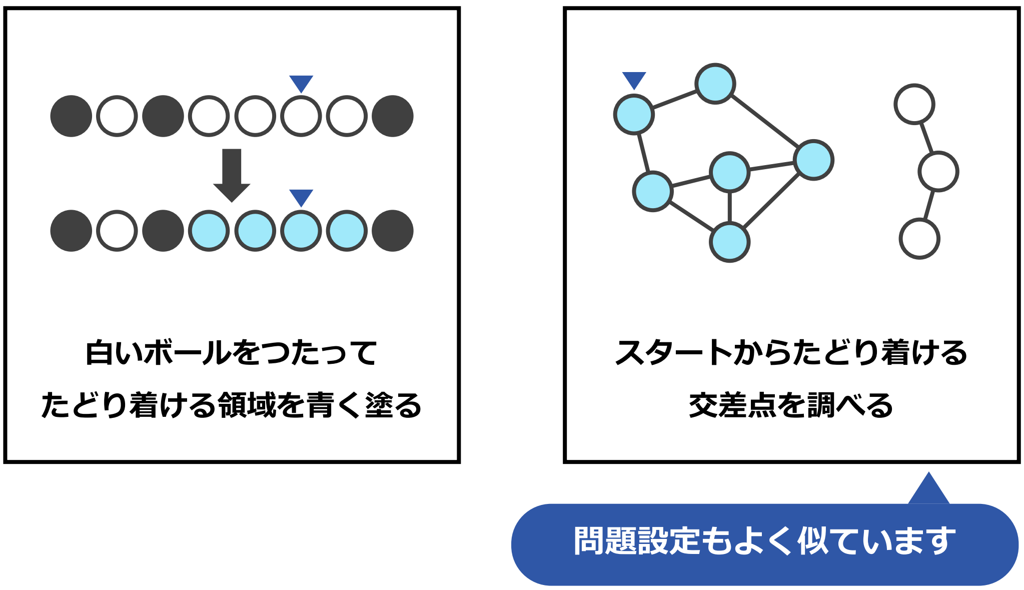


幅優先探索との関連

　今回行ったシミュレーションは、9.3 節で学ぶ「幅優先探索」と非常によく似ています。幅優先探索を使うと、

- 道路網で、あるスタート地点からたどり着ける交差点はどこか
- 最小何本の道路を通ることで、スタートからゴールへたどり着けるか

などの様々な実用的な問題を解くことができます。興味のある方は、ぜひ本の 357～361 ページをご覧ください。



8.3 問題 B53 (B39)：Taro’s Job (難易度：★4相当)

※この問題は例題と比べて相当難しいです。

応用問題 6.4 の解説に書いた通り、この問題は「今選べる中で最も給料の高い仕事を選び続ける」という貪欲法で解くことができます。

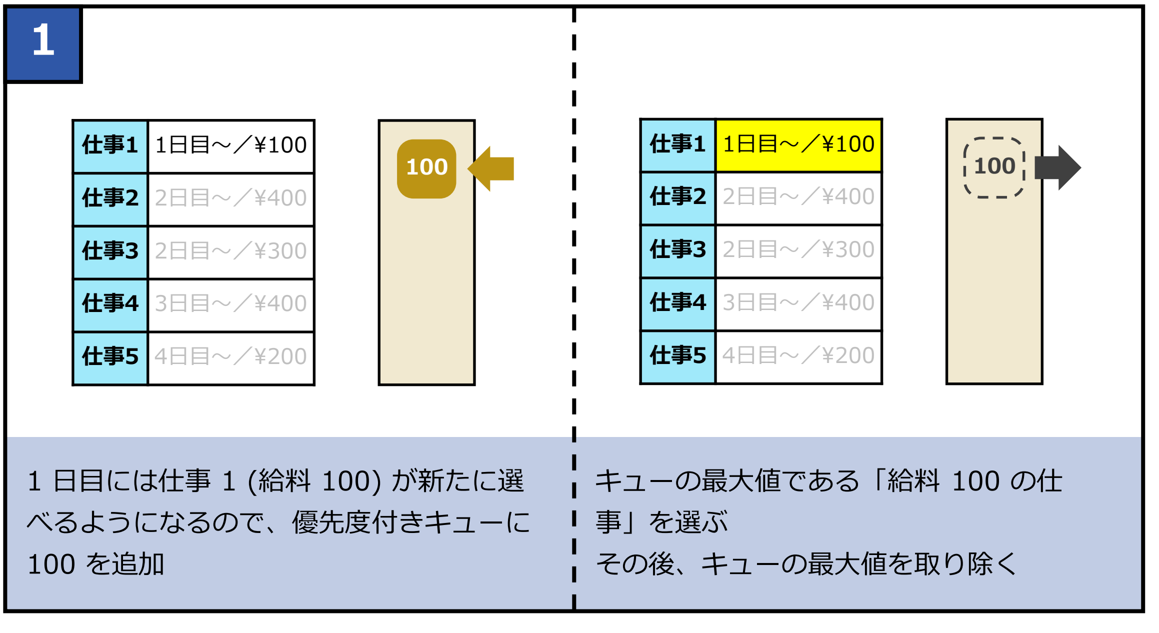
しかし、この貪欲法を自然に実装すると、1 日の仕事を選ぶのに計算量 $O(N)$ かかります。日数は D 日なので、全体の計算量は $O(ND)$ となり、残念ながら実行時間制限に間に合いません。

工夫した解法

そこで、1 日の仕事を計算量 $O(\log N)$ で選べるようにするため、**「今選べる仕事の給料のリスト」を優先度付きキューで管理する**ことを考えます。

たとえば、 $N=5, D=4, (X_i, Y_i) = (1,100), (2,400), (2,300), (3,400), (4,200)$ の場合は以下のようになります。

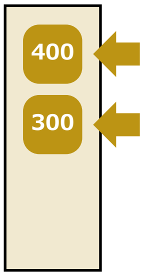
なお、使う優先度付きキューは、最大の要素を取り出すものであることに注意してください（つまり取り出す要素が “選ぶべき仕事” になります）。



※表では、選べる仕事を黒字で、選べない仕事を灰色で示しています

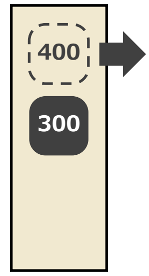
2

仕事1	完了
仕事2	2日目～／¥400
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200



2 日目には仕事 2 (給料 400) と仕事 3 (給料 300) が新たに選べるようになるので、優先度付きキューに 400, 300 を追加

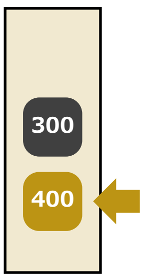
仕事1	完了
仕事2	2日目～／¥400
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200



キューの最大値である「給料 400 の仕事」を選ぶ
その後、キューの最大値を取り除く

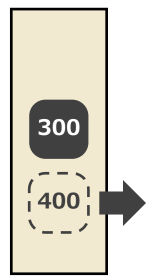
3

仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200



3 日目には仕事 4 (給料 400) が新たに選べるようになるので、優先度付きキューに 400 を追加

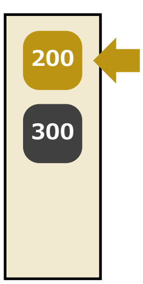
仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200



キューの最大値である「給料 400 の仕事」を選ぶ
その後、キューの最大値を取り除く

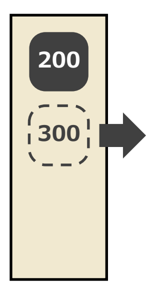
4

仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	完了
仕事5	4日目～／¥200



4 日目には仕事 5 (給料 200) が新たに選べるようになるので、優先度付きキューに 200 を追加

仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	完了
仕事5	4日目～／¥200



キューの最大値である「給料 300 の仕事」を選ぶ
その後、キューの最大値を取り除く

計算量について

最後に、計算量はどれくらいになるのでしょうか。優先度付きキューへの追加は仕事の個数と同じ N 回行いますが、削除は日数と同じ D 回行うので、優先度付きキューに対して行う操作の回数は合計 $N+D$ 回となります。

したがって、アルゴリズム全体の計算量は $O((N+D)\log N)$ です。以下に実装例を示します。

解答例（C++）

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

long long N, D;
long long X[200009], Y[200009];
vector<long long> G[200009]; // G[i] は i 日目から始まる仕事の給料のリスト
long long Answer = 0;

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

    // 答えを求める
    priority_queue<long long> Q;
    for (int i = 1; i <= D; i++) {
        // i 日目から始まる仕事をキューに追加
        for (int j : G[i]) Q.push(j);

        // やる仕事を選択し、その仕事をキューから削除する
        if (!Q.empty()) {
                Answer += Q.top();
                Q.pop();
        }
    }

    // 出力
    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

8.4 問題 B54：Counting Same Values（難易度：★2相当）

この問題は、以下のようなアルゴリズムで解くことができます。計算量は $O(N \log N)$ です。

各要素が現時点で何回出現したかを管理する連想配列 Map（初期値 0）を用意し、$i = 1,2 \ldots, N$ の順に次の処理を行う：

- **手順1：** 答え Answer に Map[X[i]] を加算する
- **手順2：** Map[X[i]] に 1 を加算する

このアルゴリズムが上手くいく理由は、**手順 1 で加算される Map[X[i]] の値が、$X_j = X_i$ $(j < i)$ を満たす j の個数である**からです。

解答例（C++）

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

int N, A[100009];
map<int, int> Map;

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

    // 答えを求める
    long long Answer = 0;
    for (int i = 1; i <= N; i++) {
        Answer += Map[A[i]];
        Map[A[i]] += 1;
    }

    // 出力
    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

8.5 問題 B55：Difference （難易度：★4相当）

まず、「差の最小値」を求めるクエリ 2 に答えるためには、以下の 2 つの値が分かっている必要があります。

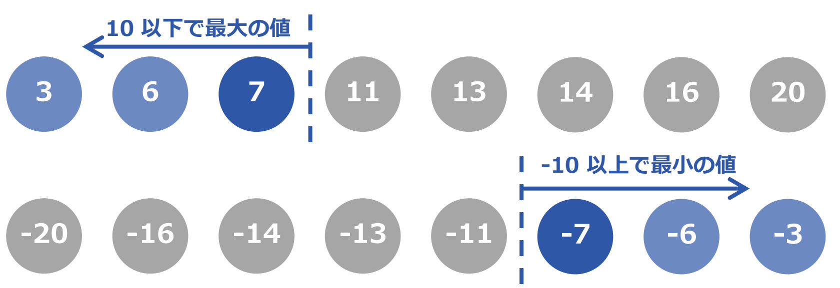
- 机にある x 以下のカードのうち、最大の値 v1
- 机にある x 以上のカードのうち、最小の値 v2

ここで、v2 の値は例題 A55（8.5 節）で学んだ通り、lower_bound 関数を使えば一発で求められます。しかし、v1 についてはどうでしょうか。

v1 を求める方法

解決策の一つとして[※1]、**「カードの値×(-1)」を記録した set を用意する**方法があります。

この方法を使った場合、**set における「$-x$ 以上で最小の値」が「x 以下で最大の値」に対応する**ので、v2 の値と同じようにして v1 の値を求めることができます。実装例を以下に示します。



解答例（C++）

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

long long N, QueryType[100009], x[100009];
set<long long> Set1, Set2;
```

※1　他にも、itr++ や itr-- などの高度な機能（本の 312 ページで解説）を使う手段もあります

```cpp
// r 以下の最大値を返す
long long GetDown(long long r) {
    auto itr = Set2.lower_bound(-r);

    // r 以下のものが存在しない場合、非常に小さい値を返す
    if (itr == Set2.end()) return -1000000000000000LL;

    // 存在する場合
    return -(*itr);
}

// r 以上の最小値を返す
long long GetUp(long long r) {
    auto itr = Set1.lower_bound(r);

    // r 以上のものが存在しない場合、非常に大きい値を返す
    if (itr == Set1.end()) return 1000000000000000LL;

    // 存在する場合
    return (*itr);
}

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

    // クエリの処理
    for (int i = 1; i <= N; i++) {
        if (QueryType[i] == 1) {
                Set1.insert(x[i]);
                Set2.insert(-x[i]);
        }
        if (QueryType[i] == 2) {
                long long v1 = GetDown(x[i]);
                long long v2 = GetUp(x[i]);
                long long Answer = min(x[i] - v1, v2 - x[i]);
                if (Answer >= 100000000000LL) cout << "-1" << endl;
                else 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

8.6 問題 B56：Palindrome Queries (難易度：★5相当)

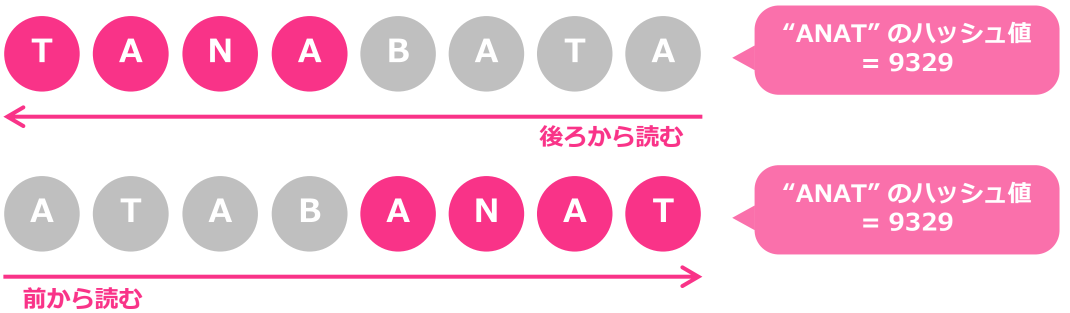
まず、回文である条件は「前から読んでも後ろから読んでも文字列が同じこと」です。

そのため、もし **$S[l,r]$ を前から順に読んだときのハッシュ値**と、**$S[l,r]$ を後ろから読んだときのハッシュ値**が同じである場合、$S[l,r]$ はほぼ 100% 回文であると分かります。

後ろからのハッシュ値はどう求める？

そこで、文字列 S の長さを N とし、文字列 S を逆順にした文字列を S' とするとき、以下の 2 つの値は一致します。

- $S[l,r]$ を**後ろから**読んだときのハッシュ値
- $S'[N-r+1, N-l+1]$ を**前から**読んだときのハッシュ値



したがって、以下のようなプログラムにより、回文かどうかを判定するクエリを処理することができます。

なお、プログラム中の関数 `GetHashLeft(l, r)` は、$S[l,r]$ を前から読んだときのハッシュ値を返します。また、プログラム中の関数 `GetHashRight(l, r)` は、$S[l,r]$ を後ろから読んだときのハッシュ値を返します。

解答例（C++）

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

// 入力で与えられる変数など
int N, Q, L[100009], R[100009];
string S;
string SRev; // S の逆順

// 文字列を数値に変換した値（それぞれ S, SRev に対応）
int T[100009];
int TRev[100009];

// ハッシュ値など
long long mod = 2147483647;
long long Power100[100009];
long long H[100009];    // S のハッシュ
long long HRev[100009]; // SRev のハッシュ

// 文字列の l～r 番目を前から読んだ時のハッシュ値を返す関数
long long GetHashLeft(int l, int r) {
    long long val = H[r] - (Power100[r - l + 1] * H[l - 1] % mod);
    if (val < 0) val += mod;
    return val;
}

// 文字列の l～r 番目を後ろから読んだ時のハッシュ値を返す関数
long long GetHashRight(int l, int r) {
    int true_l = N + 1 - r;
    int true_r = N + 1 - l;
    long long val = HRev[true_r] - (Power100[true_r - true_l + 1] * HRev[true_l - 1] % mod);
    if (val < 0) val += mod;
    return val;
}

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

```cpp
    SRev = S;
    reverse(SRev.begin(), SRev.end());

    // S, SRev の文字を数値に変換
    for (int i = 1; i <= N; i++) T[i] = (int)(S[i - 1] - 'a') + 1;
    for (int i = 1; i <= N; i++) TRev[i] = (int)(SRev[i - 1] - 'a') + 1;

    // 100 の n 乗を前計算
    Power100[0] = 1;
    for (int i = 1; i <= N; i++) Power100[i] = (100LL * Power100[i - 1]) % mod;

    // S のハッシュ値を前計算
    H[0] = 1;
    for (int i = 1; i <= N; i++) H[i] = (100LL * H[i - 1] + T[i]) % mod;

    // SRev のハッシュ値を前計算
    HRev[0] = 1;
    for (int i = 1; i <= N; i++) HRev[i] = (100LL * HRev[i - 1] + TRev[i]) % mod;

    // クエリの処理
    for (int i = 1; i <= Q; i++) {
        long long v1 = GetHashLeft(L[i], R[i]);
        long long v2 = GetHashRight(L[i], R[i]);
        // 左から読んだ時・右から読んだ時のハッシュ値が一致していれば回文
        if (v1 == v2) cout << "Yes" << endl;
        else cout << "No" << endl;
    }
    return 0;
}
```

※Python のコードはサポートページをご覧ください

8.7 問題 B57：Calculator (難易度：★5相当)

この問題も例題 A57（8.7 節）と同じように、以下の値を前計算する「ダブリング」を使うことで効率的に解けます。

- 整数 i から 1 回操作した後の整数 $\mathrm{dp}[0][i]$
- 整数 i から 2 回操作した後の整数 $\mathrm{dp}[1][i]$
- 整数 i から 4 回操作した後の整数 $\mathrm{dp}[2][i]$
- 整数 i から 8 回操作した後の整数 $\mathrm{dp}[3][i]$
- 16, 32, 64,... 回操作した後も同様

それでは、このアイデアはどうやって実装すれば良いのでしょうか。

ダブリングの実装 (1)

まずは前計算について考えます。2^{d-1} 回後の 2^{d-1} 回後は 2^d 回後ですので、前計算の部分は以下のようにして実装できます。

ここで、本問題の制約は $K \leq 10^9$ であり、$10^9 < 2^{30}$ ですので、$\mathrm{dp}[29][j]$ までを前計算しておけば十分です。

```
for (int i = 1; i <= N; i++) dp[0][i] = i - (i の各桁の和)

for (int d = 1; d <= 29; d++) {
    for (int i = 1; i <= N; i++) dp[d][i] = dp[d - 1][dp[d - 1][i]];
}
```

ダブリングの実装 (2)

次に、各整数に対して「K 回の操作を行った後の値」を求める部分について考えます。

本の 322 ページでも述べた通り、**K を 2 進法で表したときの 2^d の位が 1 であるときに限り「操作を 2^d 回進めること」を行えば、操作を K 回進めることができる**ため、この部分は次ページのように実装できます。

```cpp
// 答えを求める
for (int i = 1; i <= N; i++) {
    int CurrentNum = i; // 現在の整数
    for (int d = 29; d >= 0; d--) {
        if ((K / (1 << d)) % 2 != 0) CurrentNum = dp[d][CurrentNum];
    }
    cout << CurrentNum << endl;
}
```

最後に、以上の内容をまとめると、解答例のようになります。プログラム全体の計算量は $O(N \log K)$ です。

解答例（C++）

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

int N, K;
int dp[32][300009];

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

        // 1 回操作した後の値を求める
        for (int i = 1; i <= N; i++) {
                string str = to_string(i);
                dp[0][i] = i;
                for (int j = 0; j < str.size(); j++) {
                        dp[0][i] -= (int)(str[j] - '0');
                }
        }

        // 前計算
        for (int d = 1; d <= 29; d++) {
                for (int i = 1; i <= N; i++) dp[d][i] = dp[d - 1][dp[d - 1][i]];
        }

        // 答えを求める
        for (int i = 1; i <= N; i++) {
                int CurrentNum = i; // 現在の整数
                for (int d = 29; d >= 0; d--) {
                        if ((K / (1 << d)) % 2 != 0) CurrentNum = dp[d][CurrentNum];
                }
                cout << CurrentNum << endl;
        }
        return 0;
}
```

※Python のコードはサポートページをご覧ください

8.8 問題 B58：Jumping

（難易度：★6相当）

※この問題は例題と比べて相当難しいです。

この問題の解説を見る前に、まずは問題 A76（例題 10.6）を解くことをおすすめします。問題 B58 と非常によく似た問題ですが、難易度は ★4 程度と比較的簡単です。
なお、本を最初から順番に読んでいる方は、10 章まで読破してからこの問題に挑戦することを推奨します。

まず考えられる方法は、スタートからゴールまで移動する方法を全探索することです。しかし、移動方法は最大で 2^{N-2} 通りあるため、制約の上限である $N = 100000$ はもちろん、$N = 100$ でも絶望的です。

動的計画法を考える

そこで、以下の配列に動的計画法を適用させることを考えます。

dp[i]：足場 1 から足場 i まで最小何回のジャンプで移動できるか？

まず初期状態は明らかに dp[1]=0 です（スタート地点は足場 1 であるため）。次に状態遷移を考えます。

- posL： $X_{\mathrm{pos}} \geq X_i - R$ を満たす最小の pos
- posR： $X_{\mathrm{pos}} \leq X_i - L$ を満たす最大の pos

とするとき、足場 i にたどり着くための最後の行動としては「足場 posL, posL+1, ..., posR のいずれかから直接ジャンプする」しかありません。そのため、dp[i] の値は以下のとおりになります。

```
dp[i]=min(dp[posL], dp[posL+1], ..., dp[posR]) + 1
```

したがって、次ページに示すようなプログラムにより、ゴール地点までの最小ジャンプ回数が分かります。計算量は $O(N^2)$ です。なお、posL や posR の値は、配列の二分探索によって求めることができます。

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

int N, L, R, X[100009];
int dp[100009];

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

    // 動的計画法
    dp[1] = 0;
    for (int i = 2; i <= N; i++) {
        int posL = lower_bound(X + 1, X + N + 1, X[i] - R) - X;
        int posR = lower_bound(X + 1, X + N + 1, X[i] - L + 1) - X - 1;

        // dp[posL] から dp[posR] までの最大値を求める
        dp[i] = 1000000000;
        for (int j = posL; j <= posR; j++) dp[i] = min(dp[i], dp[j] + 1);
    }

    // 答えを出力
    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

動的計画法の高速化

先程のプログラムはたしかに正しい答えを出力します。しかし、dp[posL] から dp[posR] までの区間の最大値を求める部分がボトルネックとなり、計算量が $O(N^2)$ になってしまいます。本問題の制約は $N \leq 100000$ ですので、このままでは実行時間制限に間に合いません。

そこで、dp[i] の値をセグメント木で管理すると、区間の最大値を計算量 $O(\log N)$ で求めることができ、プログラム全体の計算量が $O(N \log N)$ まで削減されます。以下に実装例を示します。

解答例（C++）

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

class SegmentTree {
public:
    int dat[300000], siz = 1;
```

```cpp
    // 要素 dat の初期化を行う（最初は全部ゼロ）
    void init(int N) {
        siz = 1;
        while (siz < N) siz *= 2;
        for (int i = 1; i < siz * 2; i++) dat[i] = 0;
    }

    // クエリ 1 に対する処理
    void update(int pos, int x) {
        pos = pos + siz - 1;
        dat[pos] = x;
        while (pos >= 2) {
                pos /= 2;
                dat[pos] = min(dat[pos * 2], dat[pos * 2 + 1]);
        }
    }

    // クエリ 2 に対する処理
    // u は現在のセル番号、[a, b) はセルに対応する半開区間、[l, r) は求めたい半開区間
    int query(int l, int r, int a, int b, int u) {
        if (r <= a || b <= l) return 1000000000; // 一切含まれない場合
        if (l <= a && b <= r) return dat[u]; // 完全に含まれる場合
        int m = (a + b) / 2;
        int AnswerL = query(l, r, a, m, u * 2);
        int AnswerR = query(l, r, m, b, u * 2 + 1);
        return min(AnswerL, AnswerR);
    }
};

int N, L, R, X[100009];
int dp[100009];
SegmentTree Z;

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

    // セグメント木の準備
    Z.init(N);
    dp[1] = 0;
    Z.update(1, 0);

    // 動的計画法
    for (int i = 2; i <= N; i++) {
        int posL = lower_bound(X + 1, X + N + 1, X[i] - R) - X;
        int posR = lower_bound(X + 1, X + N + 1, X[i] - L + 1) - X - 1;
        dp[i] = Z.query(posL, posR + 1, 1, Z.siz + 1, 1) + 1;
        Z.update(i, dp[i]);
    }

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

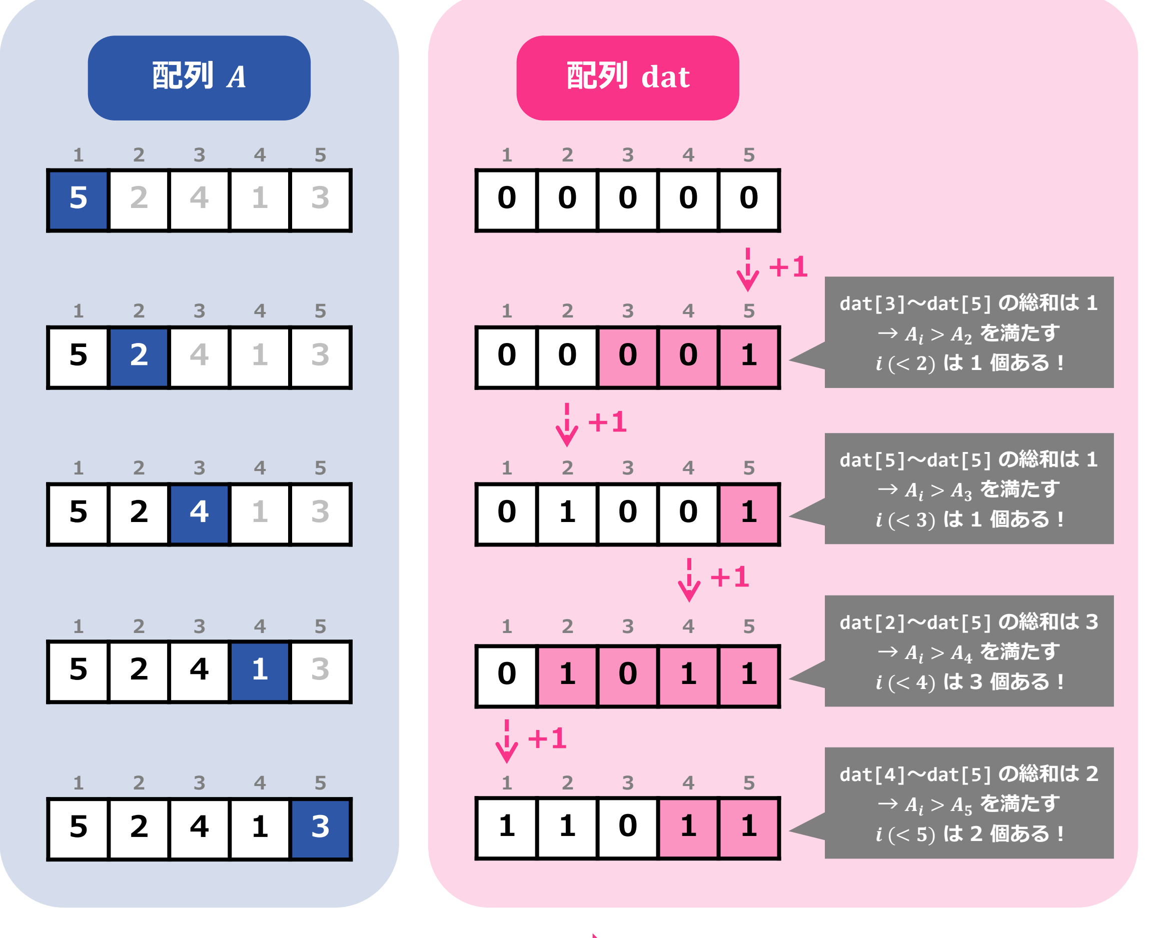
※Python のコードはサポートページをご覧ください

8.9 問題 B59：Number of Inversions（難易度：★5相当）

この問題を解く最も単純な方法は、すべての組 $(i, j)[1 \leq i < j \leq N]$ について $A_i > A_j$ かどうかを直接調べることです。しかし、この方法では計算量が $O(N^2)$ となり、残念ながら実行時間制限に間に合いません。

工夫した解法

そこで、**現時点で整数 x は何回出現したかを表す配列 dat[x]** を管理することを考えます。すると、下図のようにして $A_i > A_j$ となる組 (i, j) の個数を計算することができます（$A = [5, 2, 4, 1, 3]$ のときの例を示しています）。



答えは 1+1+3+2=7 個！

この解法を厳密に書き表すと、次のようになります。

最初は dat[1], dat[2], …, dat[N]=0 に初期化する。その後、$j = 1, \dots, N$ の順に以下の処理を行う。

- 答え Answer に dat[A[j]+1]+･･･+dat[N] を加算する
- dat[A[j]] に 1 を加算する

さて、この解法の計算量はどれくらいなのでしょうか。dat[A[j]+1] から dat[N] までの区間の総和を直接計算すると、計算量は $O(N^2)$ となり、元々の解法と変わりません。

しかし、dat[i] の値をセグメント木を使って管理すると、区間の総和を計算量 $O(\log N)$ で求めることができるため、全体の計算量は見事に $O(N \log N)$ まで削減されます。

最後に、以上の解法を実装すると、解答例のようになります。

解答例（C++）

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

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

class SegmentTree {
public:
    int dat[600000], siz = 1;

    // 要素 dat の初期化を行う（最初は全部ゼロ）
    void init(int N) {
        siz = 1;
        while (siz < N) siz *= 2;
        for (int i = 1; i < siz * 2; i++) dat[i] = 0;
    }

    // クエリ 1 に対する処理
    void update(int pos, int x) {
        pos = pos + siz - 1;
        dat[pos] = x;
        while (pos >= 2) {
                pos /= 2;
                dat[pos] = dat[pos * 2] + dat[pos * 2 + 1]; // 8.8 節から変更した部分
        }
    }
```

```
    // クエリ 2 に対する処理
    int query(int l, int r, int a, int b, int u) {
        if (r <= a || b <= l) return 0; // 8.8 節から変更した部分
        if (l <= a && b <= r) return dat[u];
        int m = (a + b) / 2;
        int AnswerL = query(l, r, a, m, u * 2);
        int AnswerR = query(l, r, m, b, u * 2 + 1);
        return AnswerL + AnswerR; // 8.8 節から変更した部分
    }
};

int N, A[150009];
SegmentTree Z;

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

    // セグメント木の準備
    Z.init(N);

    // 答えを求める
    long long Answer = 0;
    for (int j = 1; j <= N; j++) {
        Answer += Z.query(A[j] + 1, N + 1, 1, Z.siz + 1, 1);
        Z.update(A[j], 1);
    }

    // 出力
    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