第6回 日本情報オリンピック本選 解説[1]

2007年12月20日

情報オリンピック日本委員会

[1]Copyright ©2007 The Japanese Committee for International Olympiad in Informatics
著作権は情報オリンピック日本委員会に帰属します.

1　最大の和

解説

本問題は，数列の部分和に関するものである．データ構造とアルゴリズムの観点からは，連続した配列の一部である部分配列でその和が最大になるものを求める問題ととらえることができる．これを解くには，与えられた数列から作り出される長さ k の部分配列全てについて，その和を比較する必要がある．長さ k の部分配列の和を順次求めていく方法で，単純な方法（解法１）と少し工夫して効率的な方法（解法２）が考えられる．分かりやすさのために，数列に対応する配列の添え字を 1 からとし，$a[1], a[2], \ldots, a[n]$ として説明します．

解法１

単純に長さ k の部分配列の和を順に求めていき，その中で最大値を求める．最初に $a[1] + a[2] + \ddots + a[k]$ を求め，この値を MAX とする．次に $a[2] + a[3] + \ddots + a[k+1]$ を求め，この値が MAX より大きければ，MAX をこの値に更新する．以下，順に同様なことを，$a[n-k+1] + a[n-k+2] + \ddots + a[n]$ まで繰り返して，最終的に得られる MAX が求めるものである．

解法２

単純に長さ k の部分配列の和を順に求めていくが，和を求めるところで前の計算結果を利用する．最初に $a[1]+a[2]+\ddots+a[k]$ を求め，この値を MAX とし，さらに PRE とする．次に $a[2]+a[3]+\ddots+a[k+1]$ を求めるが，このとき，$a[2]+a[3]+\cdots+a[k+1] = (a[1]+a[2]+\cdots+a[k])-a[1]+a[k+1] = \mathtt{PRE}-a[1]+a[k+1]$ より，$\mathtt{PRE}-a[1]+a[k+1]$ の計算で部分配列の和を求め，この値を PRE とし，この値が MAX より大きければ，MAX をこの値に更新する．以下，同様に $\mathtt{PRE} - a[n-k] + a[n]$ まで繰り返して，最終的に得られる MAX が求めるものである．

解析

解法１の足し算の回数は，$(n-k+1)(k-1)$ となる．n と k が十分に大きいときは，大体 $(n-k)k$ となる．n を固定したとき，$k = n/2$ のとき最大となり（厳密ではないが，大体）$n^2/4$ となる．このようなことから，k の制限によるが，このアル

ゴリズムは $O(n^2)$（オーダ n 二乗）と捉えることもできる（一般には，n と k で $O(nk)$ と表しますが）.

解法 2 の足し算と引き算の回数は，$k-1+(n-k)\times 2 = 2n-k+1$ となる．これは，$O(n)$ のアルゴリズムになる.

例えば，$n=100000, k=50000$ のとき，解法 1 では，約 25 億回，解法 2 では，約 15 万回の演算が実行されることになる.

解法 1 では 60%の得点を得られ，解法 2 では 100%の得点を得られるよう，本選の制限時間と採点用入力データを調整した.

2

最長の階段

解説

アルゴリズムの方針

データを読み込んだ後,

1. 読み込んだデータを小さい順に並び替える（ソートする）
2. 並んだデータを順に見ていき、最長の連続した整数列を発見する。

という 2 ステップを行えば，高速に解を求めることができる.

データの並び替え

ここではバケットソートを簡単に紹介する.

1. サイズ $n+1$ の整数型の配列 `data` を作成し，すべて配列要素を値 0 で初期化する（バケットと呼ぶ）.
2. 以下を，カードの枚数回繰り返す.
   (a) カードに書かれている値を 1 つ読み取る．この値を j とする.
   (b) 配列の j 番目に 1 を格納する.

もし，$n=6$, $k=4$ で，カードに書かれている値が $1,5,4,3$ だった場合，上記実行後の配列 `data` は,

```
data[0] = 0
data[1] = 1
data[2] = 0
data[3] = 1
data[4] = 1
data[5] = 1
data[6] = 0
```

となる.

つまり，カードの値が書かれた容器（バケット）に中身があるか否かを，1 と 0 で表現する．連続しているバケットに 1 が入っているものが，連続した整数列となる．すなわち，この問題を解くには，連続したバケットに 1 が入っている部分のうち，一番長いものを探せば良い．上記の例だと，連続して 1 が入っている data[3],data[4],data[5] が，最長の整数列となる.

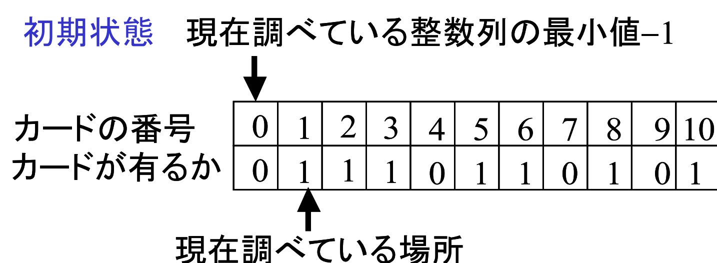
連続した整数列の発見

与えられたカードに，0 が書かれたものがない場合

ステップ 1:

次の 3 つ値を記憶する．

- 現在の最長の整数列の長さ（初期状態は 0）
- 現在調べている（連続した）整数列の最小値 −1（初期状態は 0）
  すなわち，連続した整数列の左端の 0 の場所
- 現在調べている場所（初期状態は 1）



ステップ 2:

現在調べている場所のバケットの中身が 1 である間（すなわち，その値が書かれたカードが存在している間），現在調べている場所（バケット）を，カードの番号が小さい方から大きい方へ動かしていく．

ステップ 3:

バケットの中身が 0 だった場合（すなわち，現在調べているバケットに対応するカードが無い場合），

$$\text{現在調べている場所} - (\text{現在調べている整数列の最小値} - 1) - 1$$

が今調べている整数列の長さとなる．この値が現在の最長の整数列の長さより大きいか否かを比較し，大きかったら現在の最長の整数列の長さの値を更新する．

ステップ 4:

その後，現在調べている整数列の最小値を現在調べている場所（に対応するカードの番号）と同じにし，現在調べている場所を 1 つ大きい方にずらす．

ステップ 5:

ステップ 2 に戻って，同様に行う．すべて調べ終わっていたら，最長の整数列の長さを出力して終了する．

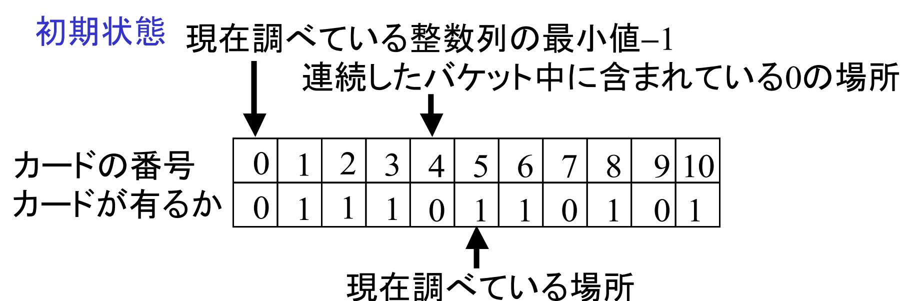
与えられたカードに，0 が書かれたものがある場合

0 が書かれたカードが与えらない場合は，連続しているバケットに 1 が入っているものが連続した整数列となる．しかし，0 が書かれたものが与えられている場合は，連続しているバケットのうち，1 つだけバケットに 0 が入っていても良いことになる．

ステップ 1:

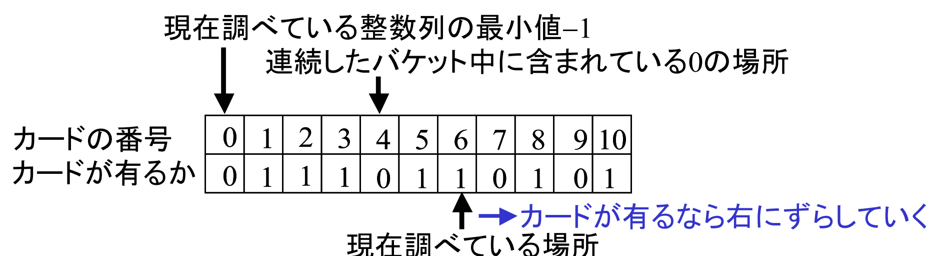
次の 4 つの値を記憶する．

- 現在の最長の整数列の長さ（初期状態は 0）
- 現在調べている整数列の最小値 −1（初期状態は 0）
- 連続したバケット中に含まれている 0 の場所（初期状態は，一番最初に無いカードの番号）
- 現在調べている場所（初期状態は、一番最初に無いカードの番号 +1）



ステップ 2:

現在調べている場所のバケットの中身が 1 である間，現在調べている場所を，カードの番号が小さい方から大きい方へ動かしていく．

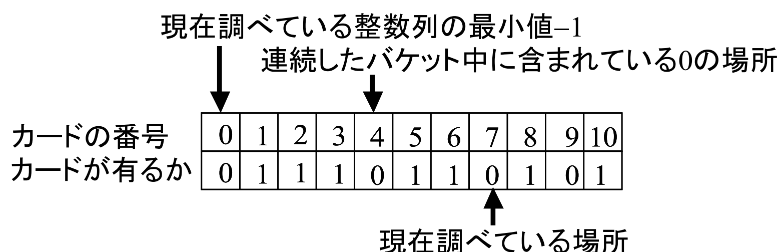


ステップ 3:

現在調べている場所の値が書かれたカードが無い場合（すなわち，バケットの中身が 0 だった場合），

$$\text{現在調べている場所} - (\text{現在調べている整数列の最小値} - 1) - 1$$

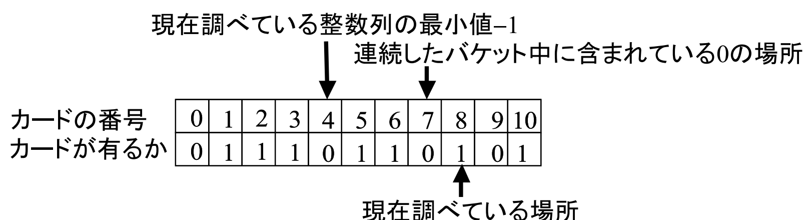
が今調べている整数列の長さとなる．この値が現在の最長の整数列の長さより大きいか否かを比較し，大きかったら現在の最長の整数列の長さの値を更新する．



ステップ 4:

その後，記憶している値を，以下のように更新する．

1. 現在調べている整数列の最小値を，連続したバケット中に含まれている 0 の場所より 1 つ大きい場所に更新する．（すなわち，現在調べている整数列の最小値 −1 が，連続したバケット中に含まれている 0 の場所になる．）
2. 連続したバケット中に含まれている 0 の場所を，現在調べている場所に更新する．
3. 現在調べている場所を 1 つ大きい方にずらす．



ステップ 5:

ステップ 2 に戻って，同様に行う．すべて調べ終わっていたら，最長の整数列の長さを出力して終了する．

3

最古の遺跡

解法

要するに, 平面上に n 個の点 $p_1, \ldots, p_n$ が与えられるので, 与えられた点でできる正方形のうち面積が最大のものを探せ, という問題である.

以下, $P = \{p_1, \ldots, p_n\}$ とし, 点 p_i の座標を (x_i, y_i) とする.

効率の良い解法

次のようにすると $O(n^2 \log n)$ 時間 $O(n)$ 領域で求めることができる.

まず, 与えられた n 個の点 $p_1, \ldots, p_n$ の座標を辞書式順序でソートしておく. こうすることで, x, y が与えられたとき, 座標 (x, y) を持つ点が P の中にあるかどうかを二分探索により $O(\log n)$ 時間で調べられる. ハッシュテーブルを使ってもよい. また, 座標が 0 から 5000 までの整数なので, ビットマップで持っておいてもよい.

次に, すべての 2 点の組 p_i, p_j $(1 \le i, j \le n,\ i \ne j)$ に対して, 線分 $p_i p_j$ を辺とする正方形であって, p_i から p_j に進む向きが正方形を反時計回りにまわるようなもの (これは 1 個に決まる) の残りの 2 頂点が P に含まれるかどうかを調べる. 具体的には, 残りの 2 頂点は

$$
\begin{aligned}
q &= (x_j - y_j + y_i, y_j + x_j - x_i), \\
r &= (x_i - y_j + y_i, y_i + x_j - x_i)
\end{aligned}
$$

なので, q と r がともに P に含まれるかどうかを調べる. また, この正方形の面積は $(x_i - x_j)^2 + (y_i - y_j)^2$ である.

以上で, P の点からなる正方形をすべて見つけることができるので, その中から面積が最大のものを見つけることもできる.

上の方法だと, 同じ正方形を必ず 4 回ずつ見つけることになる. $p_1, \ldots, p_n$ を辞書式順序でソートしてある場合は, 上の方法で $i < j$ の場合に限定してもよい (なぜなら, 各正方形について, 正方形の 4 頂点のうち添え字が最も小さいものを p_i とし, 反時計回りに見て p_i の次にくるものを p_j とおいた場合には必ず $i < j$ となるから. なお, $p_1, \ldots, p_n$ を座標の辞書式順序でソートしてある場合は, $i < j$ に限定すると各正方形をちょうど 2 回ずつ見つけることになる).

効率の悪い解法 1

4 点 p_i, p_j, p_k, p_l がこの順に反時計回りに正方形になっているかどうかは簡単なテストで $O(1)$ 時間で調べられるので, これをすべての $1 \le i, j, k, l \le n$ について試す

ことで, $O(n^4)$ 時間 $O(n)$ 領域のアルゴリズムが得られる.

この方法でもテストデータの Ex1, Ex2, 1, 2, 3 は解けるであろう. また, 既に見つかった正方形のうち最大面積のものより短い辺を無視することによって, もう少し多く解ける.

効率の悪い解法 2

「効率の良い解法」の中で, 点 q と点 r が与えられた中にあるかどうかを線形探索するようにすると, $O(n^3)$ 時間アルゴリズムが得られる.

この方法ではテストデータの Ex1, Ex2, Ex3 と 1〜6 が解けるであろう. また, 既に見つかった正方形のうち最大面積のものより短い辺を無視することによって, うまく書けばすべてのテストデータが解ける.

テストデータ

#	n	min x	max x	min y	max y	正解	正方形の個数	Remarks
Ex1	10	0	9	0	8	10	2	問題文中のサンプル
Ex2	100	266	4933	8	4983	5937209	7	サンプル入力
Ex3	500	6	4982	12	4990	5703248	9	サンプル入力
Ex4	3000	0	5000	4	5000	13442425	102	サンプル入力
1	50	21	398	0	387	3904	1	
2	100	7	4977	2	4968	0	0	
3	100	32	1929	6	2000	1382273	5	
4	500	32	4984	28	4991	13198689	26	最大正方形の辺が座標軸に平行
5	500	2	4991	1	4997	15795545	18	
6	500	57	4997	31	4950	85	38	最大正方形が 2 個
7	3000	0	1000	0	1000	417290	108	
8	3000	13	4995	2	4997	72845	65336	2 個のほぼ格子とランダムな点
9	3000	7	4999	1	4999	90	196	最大正方形が 2 個
10	3000	1	4998	0	4999	13395604	74	

4　最悪の記者

解説

不完全な情報を元に，順位表を求める問題である．この問題では，問題文を正確に理解し，適切なアルゴリズムを設計し，その正当性・効率を分析し，それを制限時間内に正しく実装する高い能力が求められている．問題文も長く，今年の本選の中では一番の難問だったようである．この問題には複数の解法が存在するが，解法によって実行時間が大きく異なるので，部分点を取ることは容易でも，満点を取るのは難しかったかもしれない．

問題は，情報に適合する順位表を一つ求めることと，それ以外に存在するかどうかを判定することの 2 つである．実は，前者ができれば，後者はやさしい．順位表を一つ求めた後で，隣り合う順位のチーム同士の試合結果が与えられているかどうかをチェックすればよい．もし各 $i = 1, \ldots, n-1$ に対して i 位のチームと $i+1$ 位のチームの勝敗が与えられていれば，順位表は 1 つのみである．与えられていない i があったら，i 位のチームと $i+1$ 位のチームを入れ替えた順位表も情報に適合するので，複数存在することが分かる．

解法 1 (効率の悪い解法)

n 個のチームの順位の付け方は全部で $n! = n \times (n-1) \times \cdots \times 2 \times 1$ 通りである．それを全て列挙して調べればよい (**全数検査法**)．これにより，計算量 $O(n!)$ の解法が得られる．

しかし，この解法は極めて効率が悪く，n がとても小さい場合にしか制限時間内に正解を出力することができないだろう．現実の問題に応用する場合は，このような解法はできるだけ避けるべきである．

解法 2 (やや効率の悪い解法)

次のように順番に順位表を求めていくことで，計算量 $O(nm)$ の解法が得られる．

まず，1 位のチームはどれかを考えよう．もしチーム a とチーム b の試合において，チーム a が勝利していたとすると，チーム b は 1 位ではない．これを与えられた m 試合の勝敗について調べることで，1 位の候補として「どのチームにも負けていないチーム」を選び出すことができる．もしそのようなチームが 1 つしかなければ，そのチームが 1 位である．2 つ以上ある場合は，どれを 1 位としてもよい．

次に，1位のチームを除いた $n-1$ 個のチームに対して同じことを行う.「1位以外にはどのチームにも負けていないチーム」を選び出す．そのようなチームが1つしかなければ，そのチームが2位であり，2つ以上ある場合はどれを2位としてもよい.

これを繰り返す．順に「1位〜$i-1$ 位のチーム以外にはどこにも負けていないチーム」を選び出し，それを i 位とおく．これにより，情報に適合する順位表が得られる.

順位表が複数存在するかどうかは次のようにして分かる．全ての i に対して,「1位〜$i-1$ 位のチーム以外にはどこにも負けていないチーム」が1つしかなければ，順位表は一意的に定まることになる．すなわち，情報に適合する順位表は1つしかない．そうでなければ複数存在する．もちろん，冒頭にも述べたように，順位表を一つ求めた後で，隣り合う順位のチーム同士の試合結果が与えられているかどうかをチェックしてもよいだろう.

各順位のチームを選び出すのに，勝敗表をチェックすると $O(m)$ 時間がかかる．従って，全体の計算量は $O(nm)$ である.

解法3 (効率の良い解法)

解法2において効率が悪い点は，勝敗表のチェックに $O(m)$ 時間がかかることである.

次のようにして，各チームが「$i-1$ 位以下のチームに何試合で負けているか」を，配列等でデータとして保持しておくことで，この部分を $O(n)$ 時間で行うことができる.

1. 下準備として，各 i に対し，番号 i のチームが負けた試合数を a_i とおく.
2. $a_i = 0$ となる i を選び，i_1 とおく．番号 i_1 のチームが勝った各チーム j に対し，a_j の値を1減らす.
3. i_1 以外で $a_i = 0$ となる i を選び，i_2 とおく．番号 i_2 のチームが勝った各チーム j に対し，a_j の値を1減らす.
4. i_1, i_2 以外で $a_i = 0$ となる i を選び，i_3 とおく．番号 i_3 のチームが勝った各チーム j に対し，a_j の値を1減らす.
5. 同様に繰り返して，数列 $i_1, i_2, \ldots, i_n$ を得る．番号 i_k のチームを k 位とする順位表が情報に適合する順位表である.

解法2と同様，途中で i_k の選び方が複数可能であれば，順位表は複数存在する．そうでなければ順位表は一意的である．もちろん，冒頭にも述べたように，順位表を一つ求めた後で，隣り合う順位のチーム同士の試合結果が与えられているかどうかをチェックしてもよい.

この解法の計算量は $O(n^2)$ である.

解法 4 (効率の良い解法)

トポロジカルソート (あるいは**深さ優先探索**) と呼ばれる手法を用いることで，情報に適合する順位表を効率よく得ることができる．

次のような関数 func を用いる．

func(i)

1. 番号 i のチームに勝っている各チーム j に対し，もし func(j) がまだ呼び出されていなかったら，func(j) を再帰的に呼び出す．
2. 番号 i を出力する．

func(i) は「チーム i に勝っているチームの番号を全て出力し，その後に自分自身の番号 i を出力する」という関数である．

プログラム内では，全ての i に対して func(i) をちょうど一度ずつ呼び出せば順位表が得られることになる (同じ func(i) を 2 回呼び出さないように，func(i) が既に呼び出されたかどうかをチェックしておく必要がある)．

この方法では情報に適合する順位表を一つ求めることしかできない．関数 func を呼び出し終わった後に，隣り合う順位のチーム同士の試合結果が与えられているかどうかをチェックして，順位表が他にあるかどうかを確かめる必要がある．

この解法の計算量は $O(n+m)$ である．

類題について

「新聞記者」という設定は，情報ではあまり馴染みがなかったかもしれない．

しかし，この問題は，以下の問題と本質的に同じ問題である．問題文の見かけに惑わされずに,「何を求めればいいのか」を考えることが大切である．

> **類題**：ある工業製品の生産ラインには 1〜n までの工程がある．いくつかの組 (i, j) に対しては,「工程 i は工程 j の前に通過しなければならない」という要請がある．いくつかの要請 (i, j) が与えられたとき，要請を全てみたすような n 個の工程の並べ方を一つ求めよ．また，そのような並べ方が，求めたもの以外にもあるかどうかを判定せよ．

今回の問題では「並べ方」(順位表) の存在は保証されていた．もちろん，さらに難しい類題として,「全ての要請をみたすように n 個の工程を並べることは可能かどうか判定せよ」というバージョンも考えられる．この問題だけでなく，いろいろなバージョンを考えてみることで，アルゴリズムを設計する能力を磨いていってほしい．

テストデータについて

サンプル入力

番号	n	m	順位表
1	4	5	1 つ
2	3	2	複数
3	7	15	1 つ
4	100	2000	複数
5	5000	100000	1 つ

採点用入力データ

番号	n	m	順位表
1	6	5	1 つ
2	7	15	1 つ
3	7	15	複数
4	30	300	複数
5	100	2000	1 つ
6	100	2000	複数
7	4999	99999	1 つ
8	4999	99999	複数
9	5000	100000	1 つ
10	5000	100000	複数

解法 1 では，全体の 30%の得点を得ることができる．また，解法 2 では，全体の 60%の得点を得ることができる．解法 3，解法 4 では満点を得ることができる．

一般的に言って，解法 4 の方が解法 3 よりも高速であるが，今回の採点用入力データでは，両者の実行時間の差はほとんどなかった．解法 3，解法 4 のどちらであっても，制限時間内に十分に余裕をもって満点を取ることができるだろう．

5

最軽量のモビール

解説

この問題の問題文のような文章を読みなれていない受験生にとってはここで言うモビールがどのようなものかを把握するのも容易くはなかったかも知れないが，錘を紐でつるしただけのものもモビールだと思えばモビール $\mathcal{M}$ の各棒の赤青の端にはモビールがつるされているということである．$\mathcal{M}$ 自身も合わせて考えればモビールの各紐 $\mathcal{S}$ 以下は１つのモビールになっている, これを $\mathcal{M}$ の $\mathcal{S}$ **の下の部分モビール**と呼ぶ. とくに, 棒の赤, 青の端についている紐の下の部分モビールをそれぞれその棒の**赤の端の部分モビール**, **青の端の部分モビール**という. また， 棒を支えている紐の下の部分もビールを単にその棒の下のモビールという.

各棒 $\mathcal{B}$ においてバランスが取れているようなモビールの重量はこのようなモビールの重量の最小値の倍数である（厳密には数学的帰納法で証明する）．モビールのある棒において支点から赤， 青の端までの長さをそれぞれ $L_{\mathrm{red}}, L_{\mathrm{blue}}$ とし，赤, 青の端の部分モビールの重量をそれぞれ $W_{\mathrm{red}}, W_{\mathrm{blue}}$ とすると $L_{\mathrm{red}}W_{\mathrm{red}} = L_{\mathrm{blue}}W_{\mathrm{blue}}$ が成り立つ．したがって, $M = L_{\mathrm{red}}W_{\mathrm{red}} = L_{\mathrm{blue}}W_{\mathrm{blue}}$ とおくと，赤， 青の端の部分モビールの重量の最小値をそれぞれ S_{red} と S_{blue} とすると M は $L_{\mathrm{red}}S_{\mathrm{red}}$ と $L_{\mathrm{blue}}S_{\mathrm{blue}}$ の公倍数であり，棒 B の下のモービルの重量が最小になるのは M が最小公倍数 M_{min} のときである．$L_{\mathrm{red}}S_{\mathrm{red}}$ と $L_{\mathrm{blue}}S_{\mathrm{blue}}$ の最小公倍数は最大公約数 d を用いて $M_{\mathrm{min}} = \left(\frac{L_{\mathrm{red}}S_{\mathrm{red}}}{d}\right) \cdot L_{\mathrm{blue}}S_{\mathrm{blue}}$ と表すことが出来るのでユークリッドのアルゴリズムを用いて最大公約数を計算すれば B 以下のモビールの重量の最小値が $\frac{M_{\mathrm{min}}}{L_{\mathrm{red}}} + \frac{M_{\mathrm{min}}}{L_{\mathrm{blue}}}$ と求まる.

後は上の計算をモビールの形状に合わせて実行して, 最も上にある棒において計算をすればモビールの重量が求まる.

上から計算

いきなり最も上のモビールにおいて上記の計算を行おうとしても棒の端についているモビールの重量が判らないと計算できないのだが, 必要な重量を必要になったら計算するという方針で上から計算する方法がある. これには再帰呼び出しという手法を使うのだが, 本選受験者でこの問題のソースファイルを提出した受験生のほとんどがこの方法で解いて（解こうとして）いた. 再帰呼び出しでモビールの重量を求める関数のアルゴリズムを以下に示す.

アルゴリズム

integer Function `Calc_Weight_Recursively`(モビール $\mathcal{M}$)
begin
  if $\mathcal{M}$ は錘である **then**
  begin
    $\mathcal{M}$ の重量 $\leftarrow 1$
  end
  else
  begin
    `RedWeight` $\leftarrow$ `Calc_Weight_Recursively`($\mathcal{M}$ の赤の端のモビール)
    `BlueWeight` $\leftarrow$ `Calc_Weight_Recursively`($\mathcal{M}$ の青の端のモビール)
    $\mathcal{M}$ の重量を計算
  end
  return $\mathcal{M}$ の重量
end

1つのモビールにあるモビールの個数は紐の本数に等しいので棒が n のとき, モビールは $2n+1$ 個ある. この計算において, 紐 $\mathcal{B}$ の下のモビールを引数として呼び出すのは $\mathcal{B}$ を支えている棒の計算をするときだけだから計算時間は $O(n)$ である.

下から計算

最も下の棒では赤, 青両端に錘がつるされている. 実際, このような棒が必ず存在することは（棒の有限性より）明らかである. 両端の部分モビールの重量がすでに判っている棒を**計算可能**な棒という. モビールにおいて計算可能な棒のいくつかで上記の計算すると新たに計算可能な棒が得られる. したがって計算可能な棒を探し, その棒において計算をし, 最も上の棒を計算するまでこれを続ければモビールの重量が求まる.

アルゴリズム

```
integer Function Calc_Weight(モビール M)
begin
    do
        foreach B in M の全ての棒
        begin
            if B は計算可能 then B の下のモビール重量を計算
            if B は最も上の棒である then break
        end
    until B は最も上の棒である
    return B の重量
end
```

最悪の場合, この計算において **do** ∼ **until** ループを実行するのに $O(n)$ 時間かかり, その間に 1 つの棒についてだけ重量を計算する. 最も上の棒が計算可能になるのはそれ以外の全ての棒を計算した後なので全体の計算時間は $O(n^2)$ である.

このままでは下からの計算は上からの計算に比べ計算時間が遅いのだが, 線形リスト（キューやスタック）を用いることにより改良できる. 要するに可変長のデータを蓄え適当な順序で取り出す仕組みがあればよい. この問題を解くには順序は考慮しないでよいのだが, 記憶した順に取り出すリストをキュー, 逆順に取り出すもをのスタックという[2]. ここでは, キューを使うこととし, 記憶を **put**, 取り出しを **get** と呼ぶ. キューを利用した $O(n)$ 時間アルゴリズムを以下に示す.

アルゴリズム

```
integer Function Calc_Weight_Que(モビール M)
begin
    foreach B in M の全ての棒
    begin
        if B は計算可能 then put(B)
    end
    do
        B = get()
        B の下のモビール重量を計算
        if B の上の棒が計算可能 then put(B の上の棒)
    until B は最も上の棒である
    return B の重量
end
```

[2]キューやスタックの性質や実装法などはセジウィック（R.Sedgewick）「アルゴリズム〈第 1 巻〉／基礎・整列」1990 年, 近代科学社を参考にすると良い

まとめ

この問題は高速なアルゴリズムを競うようなタイプのものではない. 問題を解析し, それに応じたデータ構造とそれらを取り扱うアルゴリズムを構築する問題である. 問題の要点を箇条書きにすると次のようになる.

- 1 つの棒に関する計算における数学的な考察
- 1 つの棒に関する計算の実装（ユークリッドのアルゴリズムとその応用）
- モビールの構造にあわせて計算するためのデータ構造や取り扱いの手法, 少し専門的に言えば木とその取り扱い

本選の受験生の多くはユークリッドのアルゴリズムやグラフアルゴリズムに関するある程度の知識があったのだろう. それでも, 限られた試験時間の中でこれらの問題を解決することは決して容易ではないにもかかわらずソースファイルを提出した受験生のほとんどが正しいアプローチをしていた.

この解説で紹介した 3 つのアルゴリズムのいずれを用いても問題の制限時間の範囲で計算を終了するプログラムを作成することが出来るが, おそらく実際のプログラムで試してみると実際の実行時間と理論的な考察は（小さなサイズのデータでは）必ずしも一致しないことがわかるかも知れない.