Karol Cwalina
Treść zadania, Opracowanie

Łukasz Kowalik
Program

Misie-Patysie

Bolek i Lolek bawią się w grę o swojsko brzmiącej nazwie misie-patysie. Aby zagrać w misie-patysie wystarczy mieć trochę **misiów** *oraz równie nieokreśloną liczbę* **patysiów**, *które tworzą razem* **pulę gry**. *Trzeba też wybrać dowolną liczbę naturalną, którą zgodnie z tradycją nazywa się* **MP-ograniczeniem**.

Pierwszy ruch należy do Bolka, który może zabrać z puli pewną dodatnią liczbę misiów albo patysiów. Może też jednocześnie wziąć i misie, i patysie, ale tylko jeśli jednych i drugich bierze tyle samo i więcej od zera. Oczywiście nie można zabrać więcej misiów niż jest ich w puli — podobnie z patysiami. Co więcej ani jednych, ani drugich nie można wziąć więcej niż wynosi MP-ograniczenie.

Kolejne ruchy wykonywane są na przemian według tych samych reguł. Wygrywa ten z graczy, po którego ruchu pula pozostanie pusta. Twoim celem jest pomóc Bolkowi w odniesieniu zwycięstwa nad Lolkiem.

Zadanie

Zadanie polega na napisaniu modułu, który po skompilowaniu z odpowiednim programem grającym będzie grał jako Bolek. Na potrzeby tego zadania otrzymasz **uproszczony** *program grający, który pozwoli Ci przetestować rozwiązanie. Twój moduł powinien zawierać następujące dwie procedury (funkcje):*

- **procedure** `poczatek (m, p, mpo : LongInt)` *lub*
  void `poczatek (int m, int p, int mpo)`
  Ta procedura (funkcja) będzie wywołana tylko raz, na początku rozgrywki. Możesz jej użyć by zainicjalizować zmienne lub struktury danych potrzebne podczas rozgrywki. Parametry `m`, `p` *oraz* `mpo` *zawierają odpowiednio liczbę misiów i patysiów w puli gry oraz MP-ograniczenie. Liczby Misiów i Patysiów w puli będą zawsze dodatnie i niewiększe niż 10^7. MP-ograniczenie będzie dodatnie i niewiększe niż 10^6.*

- **procedure** `ruch_bolka (m, p : LongInt; var bm, bp : LongInt)` *lub*
  void `ruch_bolka (int m, int p, int *bm, int *bp)`
  Ta procedura (funkcja) powinna wyznaczyć kolejny ruch Bolka. Przekazane jej parametry `m` *i* `p` *zawierają odpowiednio liczbę misiów i patysiów w puli gry pozostałych po ostatnim ruchu Lolka. Wartości parametrów przekazanych przy pierwszym wywołaniu procedury (funkcji)* `ruch_bolka` *są takie same jak odpowiednie wartości parametrów przekazanych do procedury (funkcji)* `poczatek`. *Przed zakończeniem wykonania procedury zmienne* `bm`, `bp` *(*`*bm` *i* `*bp` *w języku C/C++) powinny zawierać, odpowiednio, liczbę misiów i patysiów wziętych przez Bolka z puli.*

Twój moduł nie może otwierać żadnych plików ani korzystać ze standardowego wejścia/wyjścia. Jeśli twój program wykona niedozwoloną operację, w tym np. procedura `ruch_bolka` *zwróci ruch niezgodny z zasadami gry, jego działanie zostanie przerwane. W tym przypadku dostaniesz 0 punktów za dany test.*

Jeśli Twój program przegra rozgrywkę, dostaniesz 0 punktów za test. Jeśli wygra, dostaniesz maksymalną przewidzianą za dany test liczbę punktów. Dla każdego testu, na którym zostanie uruchomiony Twój program, Bolek może wygrać, niezależnie od ruchów Lolka.

Pliki (Pascal)

W katalogu `mis_pas` *znajdziesz następujące pliki:*

- `mis.pas` – *szkielet modułu grającego zawierający puste procedury* `poczatek` *i* `ruch_bolka`. *Powinieneś napisać kod tych procedur.*
- `graj.pas` – *uproszczony program generujący rozgrywkę. Ruchy Lolka wykonywane są zgodnie z bardzo prostą strategią, natomiast ruchy Bolka wyznaczane są przez wywoływanie procedur modułu grającego* `mis.pas`. *Możesz użyć tego programu do testowania swojego modułu grającego.*

Pliki (C/C++)

W katalogu `mis_c/mis_cpp` *znajdziesz następujące pliki:*

- `mis.h` – *plik zawierający nagłówki funkcji* `poczatek` *i* `ruch_bolka`.
- `mis.c/mis.cpp` – *szkielet modułu grającego zawierający puste definicje funkcji zadeklarowanych w pliku nagłówkowym* `mis.h`. *Powinieneś napisać kod tych funkcji.*
- `graj.c/graj.cpp` – *uproszczony program generujący rozgrywkę. Ruchy Lolka wykonywane są zgodnie z bardzo prostą strategią, natomiast ruchy Bolka wyznaczane są przez wywoływanie procedur modułu grającego* `mis.c/mis.cpp`. *Możesz użyć tego programu do testowania swojego modułu grającego.*

Rozwiązanie

Wynikiem Twojej pracy powinien być tylko jeden plik `mis.c`, `mis.cpp` *lub* `mis.pas`, *ale nie kilka równocześnie.*

Przykład

Rozgrywka może przebiegać następująco:

Wywołanie	*Opis*
`poczatek(7, 2, 3);`	*jest 7 Misiów, 2 Patysie, a MP-ograniczenie=3*
`ruch_bolka (7, 2, bm, bp);` *lub*	*pierwszy ruch*
`ruch_bolka (7, 2, &bm, &bp);`	*Bolek bierze z puli 1 misia i 1 patysia; zostaje 6 misiów i 1 patyś*
`ruch_bolka (3, 1, bm, bp);` *lub*	*Lolek wziął z puli 3 misie; zostały 3 misie i 1 patyś*
`ruch_bolka (3, 1, &bm, &bp);`	*Bolek bierze z puli 1 misia; zostają 2 misie i 1 patyś*
`ruch_bolka (1, 0, bm, bp);` *lub*	*Lolek wziął z puli 1 misia i 1 patysia; został 1 miś i 0 patysiów*
`ruch_bolka (1, 0, &bm, &bp);`	*Bolek bierze z puli 1 misia i wygrywa*

Rozwiązanie

Żeby z powodzeniem grać w Misie-Patysie, musimy w każdym momencie gry potrafić określić optymalny ruch. Optymalny to znaczy taki, by niezależnie od gry naszego przeciwnika, rozgrywka zakończyła się naszą wygraną. Spróbujmy więc dokonać krótkiego przeglądu metod, którymi moglibyśmy się kierować przy wyborze ruchu.

Najprostszą strategią, banalną w implementacji, jest strategia losowa. Jak jednak później pokażemy, stosujący ją gracz jest niemal pewnym przegranym w starciu z rozsądnie grającym przeciwnikiem.

Inną metodą, często spotykaną w próbach rozwiązań problemów decyzyjnych, jest poszukiwanie algorytmu zachłannego. Algorytm taki pobieżnie analizuje dostępne mu dane i wybiera ten z możliwych ruchów, który wydaje się najkorzystniejszy w aktualnej sytuacji. Przykładem zagadnienia, dla którego istnieje intuicyjny algorytm zachłanny, jest problem wydawania reszty: dysponując monetami o określonych nominałach, chcemy wydać określoną kwotę pieniędzy w jak najmniejszej liczbie monet. Zachłanna metoda postępowania polega w tym przypadku na braniu najwyższych nominałów, które nie powodują przekroczenia kwoty do wydania. Niestety w naszej grze ciężko wskazać, czym mielibyśmy się kierować przy wartościowaniu ruchów.

Istnieje jeszcze trzecia, niezawodna metoda, polegająca na przeglądaniu całego drzewa gry, czyli symulowaniu wszystkich możliwych rozgrywek i wyborze takiego ruchu, który we wszystkich rozgrywkach umożliwiał odniesienie zwycięstwa. Właściwie przy próbie analizy dowolnej popularnej gry, prędzej czy później, dochodzi się do tej metody — tak jest w przypadku szachów lub go. Być może trudna i pracochłonna analiza jest właśnie przyczyną sukcesu tych gier, gdyż gwarantują w ten sposób ciekawą i nieprzewidywalną rozgrywkę.

Przeszukiwanie drzewa gry

Jak już wspomnieliśmy, metoda ta jest niesłychanie pracochłonna, pomimo to spróbujmy zapisać odpowiedni algorytm, a następnie go zmodyfikować. Dla wygody niech μ oznacza MP-ograniczenie.

```
1: function czyWygrywajaca(m, p : longint) : boolean;
```

```
2:    { m — liczba misiów, p — liczba patysiów w puli }
3:    { true gdy zaczynający ma dla tej puli strategię wygrywającą }
4:  var i : longint;
5:  begin
6:    if m=0 and p=0 then exit(false);
7:    for i := 1 to MIN(m, μ) do
8:      if not czyWygrywajaca(m-i, p) then exit(true);
9:    for i := 1 to MIN(p, μ) do
10:     if not czyWygrywajaca(m, p-i) then exit(true);
11:   for i := 1 to MIN(m, p, μ) do
12:     if not czyWygrywajaca(m-i, p-i) then exit(true);
13:   czyWygrywajaca := false;
14: end;
15:
16: procedure ruch_bolka(m, p : longint; var bm, bp : longint);
17: var i : longint;
18: begin
19:   { próbujemy zabrać misie... }
20:   for i := 1 to MIN(m, μ) do
21:     if not czyWygrywajaca(m-i, p) then begin
22:       bm := i; bp := 0;
23:       exit;
24:     end;
25:
26:   { ...a teraz patysie... }
27:   for i := 1 to MIN(p, μ) do
28:     if not czyWygrywajaca(m, p-i) then begin
29:       bm := 0; bp := i;
30:       exit;
31:     end;
32:
33:   { ...a teraz jedne i drugie }
34:   for bm := 1 to MIN(m, p, μ) do
35:     if not czyWygrywajaca(m-bm, p-bp) then begin
36:       bm := i; bp := i;
37:       exit;
38:     end;
39:   { błąd: nie istnieje ruch wygrywający dla Bolka }
40: end;
```

Program realizujący metodę przeszukiwania całego drzewa gry został przedstawiony w pliku `mis3.pas`.

Wprawdzie tak zapisany algorytm jest poprawny, ale jego złożoność jest wykładnicza. Dzieje się tak, bo choć liczba wszystkich możliwych stanów gry wynosi $(m+1)(p+1)$, to funkcja *czyWygrywajaca* jest wywoływana bardzo wiele razy dla tych samych argumentów, a każde takie wywołanie oznacza kolejne wywołania. Taka rozrzutność jest niepotrzebna — raz

obliczone wyniki mogą zostać zapamiętane, a to już prowadzi do rozwiązania o złożoności $O(mp\mu)$.

Algorytm dynamiczny

Wspomniane usprawnienie, polegające na zapamiętywaniu wyników funkcji *czyWygrywajaca*, nosi nazwę spamiętywania i jest bardzo bliskie innej ważnej technice programowania zwanej **programowaniem dynamicznym**.

Zauważmy, że przedstawiony wyżej algorytm prowadzi do wywołań funkcji dla wszystkich możliwych par argumentów, nic nie stoi więc na przeszkodzie, byśmy w fazie inicjalizacji naszej gry zbudowali tablicę, która gromadziłaby wyniki wszystkich możliwych wywołań funkcji, a dopiero potem przystąpili do rozgrywki. W naszym przypadku odpowiedź dla kolejnych (większych) pul określamy znając wyniki dla niektórych poprzednich (mniejszych) — i to jest właśnie istota programowania dynamicznego.

Niestety, choć osiągnęliśmy już wiele, dysponujemy bardzo wolnym rozwiązaniem, o skrajnie dużych wymaganiach pamięciowych, które się nawet zwiększyły po wprowadzeniu spamiętywania/programowania dynamicznego. Musimy więc dokonać pewnych zmian w naszym algorytmie — na początku zmienimy schemat obliczeń dynamicznych.

W podejściu wzorowanym na metodzie obliczania funkcji *czyWygrywająca*, aby poznać wynik dla kolejnej puli, musimy „zebrać" rezultaty dla mniejszych pul. Tym razem spróbujmy natomiast „rozpropagować" pewien wynik już w momencie jego otrzymania. W tym celu zauważmy, że pula (m,p), składająca się z m misiów i p patysiów, jest wygrywająca jeśli dla pewnego $1 \leqslant i \leqslant \mu$ choć jedna z pul $(m-i,p)$, $(m,p-i)$, $(m-i,p-i)$ jest przegrywająca — zatem po znalezieniu puli przegrywającej (m,p) od razu możemy uznać pule $(m+i,p)$, $(m,p+i)$ i $(m+i,p+i)$, gdzie $1 \leqslant i \leqslant \mu$, za wygrywające.

To prowadzi do następującego programu, który został przedstawiony w pliku `mis2.pas`.

```
1:  var czyWygrywajaca : array [0..MaxM,0..MaxP] of boolean = {false};
2:    { MaxM, MaxP — maksymalne liczby misiów i patysiów w puli }
3:    { całą tablicę wypełniamy wartościami false }
4:
5:  procedure poczatek(m, p, μ : longint);
6:  var i, im, ip : longint;
7:  begin
8:    for im := 0 to MaxM do
9:      for ip := 0 to MaxP do
10:       if not czyWygrywajaca then begin
11:         for i := 1 to MIN(μ, MaxM-im) do
12:           czyWygrywajaca[im+i,ip] := true;
13:         for i := 1 to MIN(μ, MaxP-ip) do
14:           czyWygrywajaca[im,ip+i] := true;
15:         for i := 1 to MIN(μ, MaxM-im, MaxP-ip) do
16:           czyWygrywajaca[im+i,ip+i] := true;
17:       end;
18: end;
19:
```

```
20:  procedure ruch_bolka(m, p : longint; var bm, bp : longint);
21:    { treść tej procedury pozostaje prawie taka sama }
22:    { nawiasy po czyWygrywajaca powinny być kwadratowe, a nie okrągłe }
```

Zauważmy, że w każdej kolumnie tablicy nie więcej niż co $(\mu+1)$-sza pozycja jest przegrywająca, bo każdej takiej odpowiada μ pozycji wygrywających. Tym samym złożoność tego algorytmu wynosi $O(mp)$ i jak się przekonamy, jest on doskonałym punktem wyjścia do dalszych badań. Musimy jednak znaleźć zupełnie nowe spojrzenie na zadanie, które pozwoli nam usprawnić algorytm.

Przypadki szczególne

Bardzo często rozważenie przypadków szczególnych może nas naprowadzić na trop właściwego rozwiązania. My rozważymy dwa takie przypadki:

a) gdy pula składa się z samych Misiów;

b) gdy $\mu = \infty$, czyli innymi słowy nie ma MP-ograniczenia.

Przypadek a) jest bardzo prosty — dwóch graczy na przemian wybiera liczby naturalne ze zbioru $\{1,\ldots,\mu\}$, a zwycięzcą jest ten, który doprowadzi do tego, by suma wybranych liczb była równa m. Odrobina praktyki pozwala zauważyć, że Bolek (zaczynający gracz) jest skazany na porażkę wtedy i tylko wtedy, gdy m dzieli się przez $(\mu+1)$.

Dzieje się tak, gdyż Lolek może tak grać, by suma jego liczby i liczby wybranej chwilę wcześniej przez przeciwnika była równa $(\mu+1)$. Z drugiej strony, gdy $(\mu+1)$ nie dzieli m, to Bolek ma strategię wygrywającą, gdyż może wybrać liczbę m **mod** $(\mu+1)$ i postawić Lolka w sytuacji gracza zaczynającego, gdy nowe $m := m-(m \textbf{ mod } (\mu+1))$ dzieli się przez $(\mu+1)$.

To już coś — potrafimy dobrze grać w przypadku jednowymiarowym. Ale czy wnosi to cokolwiek do oryginalnych Misiów-Patysiów? Na szczęście tak: sugeruje nam, że czasami można swoimi ruchami dopełniać ruch przeciwnika do długości $(\mu+1)$, a w rezultacie przestać się przejmować MP-ograniczeniem.

Aby to pokazać, opiszemy najpierw grę dualną do Misiów-Patysiów, po czym na jej przykładzie pokażemy jak zapomnieć o MP-ograniczeniu. O Misiach-Patysiach można mianowicie myśleć w następujący sposób: na nieskończonej szachownicy, zajmującej pierwszą ćwiartkę układu współrzędnych, postawiono hetmana. W pojedynczym ruchu może się on przesunąć w lewo, w dół lub po skosie w lewo-dół, nie więcej jednak niż o μ. Gracze na przemian wykonują ruchy, a wygrywa ten z nich, który doprowadzi hetmana w lewy-dolny róg szachownicy.

Ta gra jest faktycznie tożsama z naszą, gdyż możemy się umówić, że wiersz i kolumna (numerowane od 0), na których stoi hetman, odpowiadają liczbie misiów i patysiów w puli w danym momencie. Ruch figury zaś odpowiada zabraniu pewnej ich liczby.

Po takim przygotowaniu pokażemy, że zachodzi następujący

Lemat 1 *Niech* $\overline{m} = m$ **mod** $(\mu+1)$, $\overline{p} = p$ **mod** $(\mu+1)$.
Wówczas $\mathit{czyWygrywająca}(m,p) = \mathit{czyWygrywająca}(\overline{m},\overline{p})$.

Dowód Myśląc o dualiźmie pomiędzy grami, podzielmy szachownicę na kwadraty o boku $(\mu+1)$ i określmy następującą strategię gry:

- gdy przeciwnik w swoim ostatnim ruchu przeniósł hetmana pomiędzy dwoma kwadratami, wykonujemy taki ruch, by w efekcie tego przesunięcia i ruchu przeciwnika zmienił się kwadrat, ale nie położenie hetmana w kwadracie;
- w przeciwnym przypadku wykonujemy taki ruch, jaki byśmy wykonali stojąc na polu $(\overline{m},\overline{p})$, ew. dowolny, jeśli stoimy w lewym-dolnym rogu kwadratu (to się może zdarzyć tylko wtedy, gdy nie będziemy zaczynać z pozycji wygrywającej).

Zauważmy, że jeśli *czyWygrywająca*$(\overline{m},\overline{p})$ = **true**, to powyższa strategia pokazuje, że *czyWygrywająca*(m,p) = **true**. Na grę możemy bowiem wtedy patrzeć jak na rozgrywkę toczoną wewnątrz pojedynczego kwadratu, której celem jest dojście do jego lewego-dolnego rogu — wystarczy pominąć te pary ruchów, które nie zmieniają położenia hetmana w kwadracie. A założyliśmy, że dla gry w kwadracie istnieje strategia wygrywająca dla zaczynającego. Podobnie można pokazać, że jeśli:

czyWygrywająca$(\overline{m},\overline{p})$ = **false**, to *czyWygrywająca*(m,p) = **false**.

Dopóki bowiem Bolek zmienia kwadrat, w którym stoi figura, Lolek wykorzystuje pierwszy punkt strategii, więc gdy pierwszy z graczy wykonuje ruchy, to stale *czyWygrywająca*$(\overline{m},\overline{p})$ = **false**; w pewnym momencie jednak Bolek musi wykonać ruch wewnątrz kwadratu, przesuwając się do pola (m',p') takiego, że *czyWygrywająca*$(\overline{m'},\overline{p'})$ = **true**, a wtedy już Lolek będzie się znajdował w sytuacji opisanej we wcześniejszym akapicie. ■

Tym samym wystarczy wyznaczenie kawałka *czyWygrywająca*[0..μ,0..μ], a w nim nie ma już sensu pojęcie MP-ograniczenia, gdyż w rozważanych pulach nie będzie nigdy więcej niż μ misiów i μ patysiów. Jednocześnie złożoność rozwiązania zmniejszyła się do $O(min(\mu^2, mp))$.

Brak MP-ograniczenia

Jak się okazało, przypadek ten w sposób jak najbardziej naturalny wypłynął w naszych rozważaniach.

Przypatrzmy się teraz sposobowi, w jaki wypełniamy tablicę *czyWygrywajaca*. Na początek przyjmijmy definicję: ***k*-tą przekątną** macierzy A nazywamy taki zbiór jej pozycji $a_{i,j}$, że $i-j=k$. Procedurę *poczatek* możemy przepisać w następującej postaci:

```
1:  procedure poczatek (m, p, μ : longint);
2:    { wypełnia pola czyWygrywająca[i,j] dla i,j ⩽ μ }
3:  var im, ip : longint
4:  begin
5:    wypełnij tablicę czyWygrywajaca wartościami UNDEF
6:    for im := 0 to μ do begin
7:      niech ip będzie najmniejsze nieujemne takie, że czyWygrywajaca[im,ip]
        jest UNDEF;
8:      czyWygrywajaca[im,ip] := false;
```

```
9:          „zaznacz” im-ty wiersz: wartości UNDEF zamień na true;
10:         „zaznacz” ip-tą kolumnę: wartości UNDEF zamień na true;
11:         „zaznacz” (im − ip)-tą przekątną: wartości UNDEF zamień na true;
12:       end;
13:   end;
```

W tym momencie jesteśmy już o krok od rozwiązania liniowego.

Przede wszystkim zauważmy, że skoro w każdej kolumnie tablicy jest tylko jedno pole z wartością **false**, moglibyśmy zrezygnować z zapisywania wartości **true** w tablicy, a zamiast tego dla każdego numeru m kolumny zapisywać numer p wiersza takiego, że $czyWygrywająca(m,p) =$ **false**. „Zaznaczenie” wiersza, kolumny lub przekątnej można by wtedy wykonać w czasie stałym.

Co więcej, ponieważ $czyWygrywająca(m,p) = czyWygrywająca(p,m)$, to w chwili znalezienia odpowiedniej pary (m,p) możemy jednocześnie „zaznaczać” pola dla przypadku (p,m). Takie postępowanie pozwala z kolei łatwo znajdować ip (7. wiersz kodu), gdyż pole (im,ip) musi leżeć na następnej przekątnej tablicy w stosunku do poprzednio znalezionej pary.

Całość zmian prowadzi do algorytmu wzorcowego o czasowej i pamięciowej złożoności $O(\mu)$.

Odpowiednie programy zapisane są w plikach `mis.pas`, `mis.c` i `mis.cpp`.

Inne rozwiązania

Wszystkie poprawne rozwiązania są prawdopodobnie mniej lub bardziej zbliżone do przedstawionego przez nas rozwiązania wzorcowego, ew. do którejś z przedstawionych wyżej jego nieefektywnych wersji.

Spośród rozwiązań niepoprawnych ciężko wybrać jakieś szczególnie interesujące — godne zauważenia jest chyba tylko, że ponieważ w każdej kolumnie i wierszu co $(\mu+1)$-sze pole jest wygrywające, to prawdopodobieństwo zwycięstwa w przypadku posługiwania się strategią losową wynosi mniej więcej $(\frac{1}{\mu})^{liczba\ ruch\acute{o}w}$, a więc jest skrajnie małe. Śmiało można wysunąć hipotezę, że równie ciężko w ten sposób zwyciężyć, co ułożyć puzzle rzucając nimi o ścianę. Aby umożliwić wszystkim chętnym wypróbowanie własnej cierpliwości, strategia losowa została zaimplementowana w pliku `misb2.pas` — nie radzimy jednak czekać na zwycięstwo tego programu dla dużych testów.

Testy

Rozwiązania zawodników były sprawdzane na zestawie 12 testów, z których `1a` i `1b` oraz `4a` i `4b` zostały zgrupowane — aby dostać punkty za grupę, należało zaliczyć oba testy z grupy.

nr testu	m	p	μ	opis
1a	2	2	5	Poprawnościowy; $p < \mu$.
1b	5	5	5	Poprawnościowy.
2	100	800	15	Wydajnościowy (odsiewa wykładnicze).
3	501	501	1	j.w.
4a	50000	20	1000	Odsiewa $O(mp)$ bez dynamicznej alokacji pamięci.
4b	20	50000	1000	j. w.
5	10^5	10^5	50	Wydajnościowy (odsiewa $O(mp)$).
6	123456	654321	98765	Wydajnościowy.
7	10^7	1	10^6	j. w.
8	99999	99999	10^6	j. w.
9	10^6	10^6	$2 \cdot 10^6$	j. w.
10	10^7	10^7	10^6	Wydajnościowy, maksymalny.

Rozwiązanie wykładnicze przechodzi testy 1a, 1b. Algorytmy o złożoności $\Theta(mp)$, bez dynamicznego przydziału pamięci — 1a, 1b, 2, 3; $\Theta(mp)$ z dynamicznym przydziałem pamięci — 1a, 1b, 2, 3, 4a, 4b; $\Theta(\mu^2)$ wszystkie testy do 5 włącznie. Testy 6-10 przechodzi jedynie rozwiązanie wzorcowe.