Tomasz Kulczyński — Treść zadania
Tomasz Kulczyński, Juliusz Sompolski — Opracowanie
Juliusz Sompolski — Program

Dostępna pamięć: 32 MB. *OI, Etap III, dzień drugi, 02.04.2009*

Poszukiwania

Młodzi i szaleńczo w sobie zakochani Bajtek i Bajtyna wpadli w niezłe tarapaty. Zły czarownik Bitocy porwał Bajtynę, mając nadzieję na sowity okup. Bajtek nie poddaje się jednak — nie jest zbyt majętny, więc postanowił odbić swoją wybrankę. Bitocy nie lubi walki wręcz, zaproponował więc inne rozwiązanie całej sytuacji. Jeśli młodzieniec zgadnie, na którym piętrze wieży znajduje się uwięziona, czarodziej puści ją wolno.

Pięter jest bardzo wiele — są ponumerowane od 1 do n. Jedyną pomocą dla Bajtka mogą być pytania zadawane czarodziejowi. Pytania muszą być postaci „Czy Bajtyna jest wyżej/niżej niż piętro x?". Oczywiście, Bajtek może wybrać dokładnie jeden z wyrazów „wyżej", „niżej", a także dowolnie ustalić liczbę x. Bitocy zawsze odpowiada na tak postawione pytanie zgodnie z prawdą, ale każe sobie zapłacić a bajtalarów, jeśli odpowiedź brzmi „tak", lub b bajtalarów, gdy odpowiedź brzmi „nie". Cóż, jeśli Bajtek zbytnio zubożeje, a Bitocy nadmiernie się wzbogaci, to Bajtyna może zechcieć zostać u czarodzieja...

Bajtek zastanawia się, jakie pytania zadawać. Niestety Bajtyna słyszy całą rozmowę, tzn. kolejne pytania Bajtka i odpowiedzi Bitocego, a jest osobą bardzo oszczędną. Jeśli tylko Bajtek wyda choćby o jednego bajtalara więcej, niż jest to (w najgorszym przypadku) niezbędne do ustalenia jej położenia, to obrazi się na niego śmiertelnie i odejdzie z Bitocym. Dokładniej, jeżeli w pewnym momencie rozmowy da się wywnioskować, że od tego momentu, niezależnie od dalszych odpowiedzi Bitocego, Bajtek może odgadnąć położenie Bajtyny, wydając przy tym nie więcej niż K bajtalarów, a od tego momentu Bajtek wyda kwotę większą niż K, to jego szanse u Bajtyny spadną do zera (punktów za dany test). Pomóż Bajtkowi!

Komunikacja

Powinieneś zaimplementować program, który rozwiąże problem Bajtka, korzystając z dostarczonej biblioteki (symulującej czarodzieja Bitocego). Aby użyć biblioteki, należy wpisać w swoim programie:

- *C/C++:* `#include "poslib.h"`
- *Pascal:* `uses poslib;`

Biblioteka udostępnia trzy procedury i funkcje:

- `inicjuj` — *podaje liczbę pięter n oraz koszty a i b. Powinna zostać użyta dokładnie raz, na samym początku działania programu.*
  - *C/C++:* `void inicjuj(int *n, int *a, int *b);`
  - *Pascal:* `procedure inicjuj(var n, a, b: longint);`
- `pytaj` — *znak c oznacza rodzaj pytania (`'W'` dla wyżej lub `'N'` dla niżej), a x to numer piętra. Wynikiem funkcji jest wartość logiczna odpowiedzi czarodzieja. Twój program może użyć tej funkcji dowolną liczbę razy.*

  - *C/C++:* `int pytaj(char c, int x);` *(0 oznacza fałsz, a 1 prawdę),*
  - *Pascal:* `function pytaj(c: char; x: longint): boolean;`

- `odpowiedz` — *za pomocą tej procedury/funkcji podajesz numer piętra, na którym jest Bajtyna. Powinna zostać użyta dokładnie raz. Jej wykonanie zakończy działanie Twojego programu.*
  - *C/C++:* `void odpowiedz(int wynik);`
  - *Pascal:* `procedure odpowiedz(wynik: longint);`

Twój program nie może otwierać żadnych plików ani używać standardowego wejścia i wyjścia. Rozwiązanie będzie kompilowane wraz z biblioteką następującymi poleceniami:

- *C:* `gcc -O2 -static poslib.c pos.c -lm`
- *C++:* `g++ -O2 -static poslib.c pos.cpp -lm`
- *Pascal:* `ppc386 -O2 -XS -Xt pos.pas`

W katalogu `/home/zawodnik/rozw/lib` *możesz znaleźć przykładowe pliki bibliotek i nieoptymalne rozwiązania ilustrujące sposób ich użycia. Aby podane powyżej polecenia kompilacji działały, pliki bibliotek powinny znajdować się w bieżącym katalogu.*

Limity

Możesz założyć, że $1 \leqslant n \leqslant 10^9$ *oraz* $1 \leqslant a,b \leqslant 10\,000$.

Przykładowy przebieg programu

Wywołanie funkcji	**Zwracane wartości i wyjaśnienia**
Pascal: `inicjuj(n,a,b);` C lub C++: `inicjuj(&n,&a,&b);`	Od tego momentu $n = 5$, $a = 1$, $b = 2$.
`pytaj('W',3);`	Wynik równa się 0/`false`. Pytasz, czy Bajtyna jest powyżej trzeciego piętra. Uzyskujesz odpowiedź, że nie. Płacisz 2 bajtalary.
`pytaj('N',2);`	Wynik równa się 0/`false`. Pytasz, czy jest poniżej drugiego piętra. Uzyskujesz odpowiedź „nie”, za którą płacisz kolejne 2 bajtalary.
`pytaj('W',2);`	Wynik równa się 1/`true`. Pytasz, czy jest powyżej drugiego piętra. Uzyskujesz odpowiedź „tak”, za którą płacisz bajtalara.
`odpowiedz(3);`	Odpowiadasz, że Bajtyna znajduje się na trzecim piętrze. Jest to poprawna odpowiedź. Wydałeś łącznie 5 bajtalarów.

Powyższy przebieg interakcji jest poprawny, ale nieoptymalny, a więc program nie uzyskałby punktów za taki test. W szczególności, dobrze napisany program potrafi dla danych $n = 5$, $a = 1$, $b = 2$ *tak zadawać pytania, żeby w każdym przypadku wydać co najwyżej 4 bajtalary.*

Rozwiązanie

Pierwsze podejście

Pierwsze rozwiązanie podchodzi do problemu wprost, obliczając dla każdego (całkowitego dodatniego) k, jakim minimalnym kosztem można znaleźć Bajtynę w przedziale zawierającym k pięter, w przypadku pesymistycznym. Zauważmy, że wspomniany minimalny koszt nie zależy od umiejscowienia tych k pięter w wieży, a jedynie od ich liczby, tzn. k. Możemy zatem oznaczyć ten koszt przez $t[k]$.

Wartości $t[k]$ będziemy obliczać w kolejności rosnących k, używając programowania dynamicznego. Oczywiście $t[1] = 0$. Jeżeli $k > 1$, to do zidentyfikowania położenia Bajtyny konieczne jest zadanie co najmniej jednego pytania; załóżmy, że jest to pytanie, czy Bajtyna znajduje się niżej niż $(p+1)$-sze od dołu piętro ($1 \leqslant p < k$). Jeżeli Bitocy odpowie, że tak, to za to pytanie Bajtek zapłaci a bajtalarów, a w dalszych pytaniach będzie mógł ograniczyć się do fragmentu wieży zawierającego tylko p pięter. W przeciwnym przypadku Bajtek poniesie koszt b bajtalarów, a przedział pięter zawęzi się do długości $k - p$. Zauważmy, że w obu przypadkach udało nam się w prosty sposób wyrazić *cały* zasób informacji, jakie Bajtek może wywnioskować z jednej odpowiedzi Bitocego.

Zakładając, że w pozostałej części konwersacji Bajtek będzie zadawał pytania w sposób minimalizujący dalszy koszt, otrzymujemy, że w przypadku odpowiedzi „tak" (na pierwsze pytanie) Bajtek wyda $a + t[p]$ bajtalarów, a w przypadku odpowiedzi „nie" — $b + t[k - p]$ bajtalarów. W takim razie pesymistyczny koszt w przypadku tego pytania to większa z tych dwóch liczb.

Zauważmy teraz, że pytania:

- „Czy Bajtyna jest wyżej niż piętro p?"
- „Czy Bajtyna jest niżej niż piętro $k - p + 1$?"

nie różnią się wcale pod względem długości przedziałów pięter uzyskanych po odpowiedzi (tj. p oraz $k - p$) oraz odpowiadających im kosztów pierwszego pytania (odpowiednio a oraz b). To pokazuje, że pytań typu „wyżej" nie musimy wcale rozważać. Aby zatem obliczyć $t[k]$, należy znaleźć takie p, by koszt zadania pytania typu „niżej" dla piętra numer $p + 1$ był minimalny. Mamy więc równość:

$$t[k] = \min_{1 \leqslant p < k} \max(\, a + t[p],\, b + t[k - p]\,).$$

Ponadto przez $p[k]$ oznaczmy wartość p realizującą to minimum, czyli spełniającą:

$$t[k] = \max(\, a + t[p[k]],\, b + t[k - p[k]]\,).$$

W ten sposób w złożoności czasowej $O(n^2)$ i pamięciowej $O(n)$ możemy wyznaczyć wszystkie wartości $t[k]$ oraz $p[k]$ dla $k = 1, 2, \ldots, n$. Po tych wstępnych obliczeniach schemat zadawania pytań jest już bardzo prosty do skonstruowania: jeżeli w danym momencie rozwiązania ograniczyliśmy rozważania do k-piętrowego fragmentu wieży ($k > 1$), to kolejnym pytaniem powinno być „Czy Bajtyna jest niżej niż piętro $p[k] + 1$?". Wówczas faza zadawania pytań będzie miała już tylko złożoność $O(n)$.

Takie rozwiązanie zostało zaimplementowane w plikach `poss1.cpp` oraz `poss1p.pas`; na zawodach przechodziło ono pierwsze trzy testy.

Poprawki

Okazuje się, że powyższe rozwiązanie można jeszcze trochę usprawnić. W tym celu należy zauważyć, że funkcja

$$f(p) = \max(\, a+t[p], b+t[k-p]\,)$$

ze wzoru

$$t[k] = \min_{1 \leqslant p < k} f(p)$$

jest najpierw nierosnąca, a potem niemalejąca. Jest tak, ponieważ $f(p)$ równa się maksimum funkcji nierosnącej $b+t[k-p]$ i funkcji niemalejącej $a+t[p]$.

Możemy więc próbować szukać jej minimum szybciej niż przez sekwencyjne sprawdzanie wartości p, a mianowicie za pomocą tzw. *wyszukiwania ternarnego*. Załóżmy, że przedział poszukiwań minimum funkcji $f(p)$ zawęziliśmy już do $p \in [l,r]$. Podzielmy przedział $[l,r]$ na trzy mniej więcej równe części w punktach $p_1 = \lfloor \frac{2l+r}{3} \rfloor$ i $p_2 = \lfloor \frac{l+2r}{3} \rfloor$. Zachodzi jeden z trzech przypadków:

- $f(p_1) < f(p_2)$ — wiemy wówczas, że p_2 jest położone w przedziale, w którym f jest niemalejąca, więc p_2 znajduje się za minimum funkcji. Możemy zatem zawęzić przedział poszukiwań do $p \in [l, p_2 - 1]$.
- $f(p_1) > f(p_2)$ — analogicznie, wówczas p_1 znajduje się przed minimum f, a więc możemy zawęzić przedział poszukiwań do $p \in [p_1 + 1, r]$.
- $f(p_1) = f(p_2) \stackrel{\text{def}}{=} m$ — w tej sytuacji nie jesteśmy w stanie łatwo stwierdzić, czy p_1 i p_2 są po lewej, po prawej, czy po przeciwnych stronach minimum. Możemy jednak próbować szukać minimum rekurencyjnie w przedziałach $[l, p_1]$, $[p_1+1, p_2-1]$ i $[p_2, r]$. Jeżeli w którymkolwiek z tych przedziałów znajdziemy minimum mniejsze niż m, to jest to na pewno minimum całej funkcji i nie musimy sprawdzać pozostałych przedziałów. Pesymistycznie jednak może zajść konieczność sprawdzenia wszystkich przedziałów.

Gdyby nie trzeci przypadek, rozwiązanie to miałoby złożoność czasową $O(n \log n)$, gdyż minimum f znajdowalibyśmy w czasie $O(\log n)$, za każdym podziałem redukując długość przedziału poszukiwań minimum o czynnik $\frac{2}{3}$. Ostatni przypadek sprawia jednak, że rozwiązanie to może działać w złożoności $O(n^2)$, choć w praktyce działa dużo szybciej niż pierwsze.

Implementacje tego rozwiązania można znaleźć w plikach `poss2.cpp` i `poss2p.pas`; przechodziły one na zawodach 5 pierwszych grup testów. W rozwiązaniach tych zastosowano pewne drobne, odkryte eksperymentalnie usprawnienia, na przykład to, że dla małych wartości parametru k szybszy w praktyce okazuje się wcześniejszy algorytm z wyszukiwaniem liniowym.

Rozwiązanie wzorcowe

Odwróćmy stawiany w zadaniu problem. Zamiast wyznaczać koszt znalezienia odpowiedzi dla danego przedziału pięter, dla każdego kosztu c obliczmy, ile co najwyżej pięter może

występować w zakresie poszukiwań, żeby koszt optymalnego wyszukania wśród tych pięter wyniósł nie więcej niż c. Oznaczmy tę szerokość przedziału (liczbę pięter) przez $Q[c]$.

Na początek zauważmy, że $Q[0] = 1$. Przyjmijmy dla wygody, że dla $c < 0$ mamy $Q[c] = -\infty$. Niech teraz $c > 0$. Załóżmy, że poszukiwania Bajtyny ograniczyliśmy już do przedziału złożonego z $Q[c]$ pięter i chcemy zadać czarownikowi następne pytanie. Po odpowiedzi przedział podzieli się na dwa krótsze, w których musimy umieć wyznaczyć rozwiązanie odpowiednio za $c-a$ i $c-b$ bajtalarów (nie wiemy, na który z nich wskaże Bitocy). Przedziały te muszą więc mieć długości nie większe niż $Q[c-a]$ i $Q[c-b]$. Stąd już prosto wnioskujemy, że:

$$Q[c] = \max(Q[c-a] + Q[c-b], 1)$$

i że do wyznaczania wartości $Q[c]$ możemy, podobnie jak w poprzednim rozwiązaniu, zastosować programowanie dynamiczne.

Aby teraz zadać odpowiednie pytanie, wystarczy znaleźć najmniejszy taki koszt w, że $Q[w-1] < n \leqslant Q[w]$. Jest to koszt, który pesymistycznie będziemy musieli ponieść. Przyjmijmy $n_0 = n$, $w_0 = w$. Kolejne nasze pytania będą dzielić przedział długości

$$n_i \leqslant Q[w_i] = Q[w_i - a] + Q[w_i - b]$$

na przedziały długości $Q[w_i - a]$ (do którego powinno zawężać się poszukiwanie po otrzymaniu odpowiedzi „tak") oraz $n_i - Q[w_i - a] \leqslant Q[w_i - b]$ (do którego powinno zawężać się poszukiwanie po otrzymaniu odpowiedzi „nie"). Wówczas po odpowiedzi Bitocego albo zapłacimy a bajtalarów i będziemy wyszukiwali w przedziale długości $n_{i+1} = Q[w_i - a]$, albo zapłacimy b bajtalarów i będziemy wyszukiwali w przedziale długości $n_{i+1} \leqslant Q[w_i - b]$. Możemy więc utrzymać się w pesymistycznym koszcie.

Zauważmy jednak, że w przypadku odpowiedzi „nie", niekoniecznie $w' = w_i - b$ jest najmniejszym takim w', że $Q[w'] \geqslant n_{i+1}$. Być może można pesymistyczny koszt w_{i+1} poprawić, próbując kolejnych, mniejszych wartości:

$$w_{i+1} = \min\left\{ j : \;\; j \leqslant w' \;\wedge\; Q[j] \geqslant n_{i+1} \right\}. \tag{1}$$

Konieczność wykonania tego kroku wynika z fragmentu treści zadania, w którym jest zdefiniowane to, jakie dokładnie wymagania odnośnie do strategii zadawania pytań stawia Bajtyna.

Rozwiązanie to ma złożoność $O(w)$ — musimy obliczyć $Q[i]$ dla wszystkich $i \leqslant w$ i przejść po nich w kierunku rosnących wartości i w trakcie obliczania w, a później — w kierunku malejących wartości podczas zadawania pytań.

```
1: inicjuj(n,a,b);
2: Q[0] := 1;
3: w := 0;
4: while Q[w] < n do begin
5:    w := w + 1;
6:    if w < a or w < b then
7:       Q[w] := 1;
8:    else
9:       Q[w] := Q[w - a] + Q[w - b];
```

```
10: end
11: dol := 1;
12: gora := n;
13: while dol < gora do begin
14:   while Q[w-1] >= gora-dol+1 do
15:     w := w-1;
16:   if pytaj('N', dol+Q[w-a]) then begin
17:     gora := dol+Q[w-a]-1;
18:     w := w-a;
19:   end
20:   else begin
21:     dol := dol+Q[w-a];
22:     w := w-b;
23:   end
24: end
25: odpowiedz(dol);
```

Aby oszacować złożoność powyższego rozwiązania względem n, należy ograniczyć z góry wartość w. Zauważmy, że ciąg $Q[i]$ jest niemalejący. Stąd dla $i \geqslant \max(a,b)$ mamy

$$Q[i] = Q[i-a] + Q[i-b] \geqslant 2Q[i-\max(a,b)].$$

Używając wielokrotnie tej nierówności, otrzymujemy, że $Q[\lceil \log_2 n \rceil \cdot \max(a,b)] \geqslant n$, a zatem

$$w \leqslant \lceil \log_2 n \rceil \cdot \max(a,b).$$

Rozwiązanie to działa więc w czasie $O(\log n \cdot \max(a,b))$ i takiej samej pamięci — gdyż musi przechowywać obliczoną tablicę Q. Jego implementacje można znaleźć w plikach `pos.cpp` oraz `pos1.pas`.

Na koniec warto wspomnieć, że w zamierzchłej, VII edycji Olimpiady Informatycznej pojawiło się zadanie interaktywne *Jajka*, którego rozwiązanie przypominało nieco rozwiązanie niniejszego zadania. Po opis tego zadania odsyłamy do książeczki owej olimpiady [7], a także na stronę `http://was.zaa.mimuw.edu.pl`.

Strategie

Do sprawdzania rozwiązań użyto biblioteki pobierającej wartości n,a,b i wykorzystującej następujące strategie:

1. Strategia `sprawiedliwa` — pobiera z wejścia numer piętra, na którym znajduje się Bajtyna, i cały czas odpowiada zgodnie z tym piętrem. W trakcie rozgrywki oszacowanie na pesymistyczny koszt może się poprawiać bądź nie, w zależności od odpowiedzi.

2. Strategia `pesymistyczna` — odpowiada tak, aby Bitocy dostał jak najwięcej pieniędzy — czyli aby pesymistyczne oszacowanie nie poprawiało się. Wbrew pozorom nie jest to najzłośliwsza strategia, gdyż przepuszcza programy, które po każdym ruchu nie sprawdzają, czy da się poprawić oszacowanie na pesymistyczny koszt zgodnie ze wzorem (1).

3. Strategia `optymistyczna` — odpowiada tak, aby po każdym pytaniu pesymistyczne oszacowanie na koszt jak najbardziej się poprawiało. Wymaga od programów zawodników, aby również poprawiały oszacowanie pesymistycznego kosztu.

Warto zaznaczyć, że dwie ostatnie z tych strategii polegają na swego rodzaju oszustwie — zamiast umieścić Bajtynę na wybranym piętrze i prawdomównie odpowiadać na pytania Bajtka, reprezentowany przez bibliotekę Bitocy decyduje o jej umiejscowieniu dopiero w trakcie rozgrywki z Bajtkiem. Zauważmy, że dopóki udzielane po drodze odpowiedzi nie są sprzeczne, czyli cały czas istnieje numer piętra zgodny z nimi wszystkimi, dopóty całe to oszustwo jest niewykrywalne dla Bajtka (a więc także dla programu reprezentującego go).

Biblioteka oblicza i aktualizuje oszacowanie na koszt tak, jak czyni to rozwiązanie wzorcowe. Po każdym pytaniu analizuje, czy program zawodnika nadal może utrzymać się w pesymistycznym oszacowaniu kosztu, a jeśli nie, od razu kończy się z wynikiem błędnym. Nie dopuszcza także strzelania odpowiedzi — analizuje, do jakiego przedziału pięter zawodnik zawęził poszukiwania, i jeśli rozwiązanie zwróci odpowiedź, zanim przedział ten zawęzi się do jednego piętra, również kończy się z wynikiem błędnym (tutaj ponownie mamy przykład niewykrywalnego oszustwa w wykonaniu biblioteki).

Testy

Testy zostały przygotowane zarówno w postaci pojedynczych przypadków testowych, jak i grup testów o tych samych n, a, b i różnych strategiach.

Nazwa	n	Opis
pos1a-1c.in	20	grupa bardzo małych testów o wszystkich możliwych strategiach; jest ona przeznaczona do przejścia przez najbardziej nieoptymalne rozwiązania, o ile rzeczywiście są poprawne (działają dla wszystkich strategii)
pos1d.in	1	przypadek skrajny
pos2.in	$\approx 8\,000$	test ze strategią `pesymistyczną` do przejścia przez rozwiązania wolniejsze oraz niektóre rozwiązania błędne
pos3a-3c.in	$\approx 8\,000$	grupa testów do przejścia przez rozwiązania wolniejsze
pos4.in	$\approx 50\,000$	test ze strategią `pesymistyczną` do przejścia przez rozwiązanie wolniejsze z optymalizacjami oraz przez niektóre rozwiązania błędne
pos5a-5c.in	$\approx 50\,000$	grupa testów do przejścia przez rozwiązanie wolniejsze z optymalizacjami
pos6.in	$\approx 1\,000\,000$	test ze strategią `pesymistyczną` do przejścia przez ewentualnie znalezione przez zawodnika szybsze rozwiązanie wolniejsze
pos7a-7c.in	$\approx 1\,000\,000$	grupa testów do przejścia przez ewentualne znalezione przez zawodnika szybsze rozwiązanie wolniejsze

Nazwa	n	Opis
pos8-9.in	$\approx 10^9$	testy ze strategią `pesymistyczną`
pos10-14a-b.in	$\approx 10^9$	testy do przejścia tylko przez rozwiązanie wzorcowe

XXI Międzynarodowa Olimpiada Informatyczna,

Plovdiv, Bułgaria 2009