Karol Pokorski
Treść zadania, Opracowanie

Karol Pokorski, Piotr Smulewicz
Program

Dostępna pamięć: 256 MB. *OI, etap I, 19.10–16.11.2015*

Korale

Bajtyna ma n korali ponumerowanych liczbami od 1 do n. Korale są parami różne. Pewne z nich są bardziej wartościowe od innych – dla każdego z korali znana jest jego wartość w bajtalarach.

Bajtyna chciałaby stworzyć naszyjnik z niektórych ze swoich korali. Jest wiele sposobów utworzenia takiego naszyjnika. Powiemy, że dwa sposoby są różne, jeśli zbiory korali użytych do ich konstrukcji są różne. Aby nieco ułatwić sobie wybór, Bajtyna postanowiła uporządkować wszystkie sposoby utworzenia naszyjnika.

Najważniejszym kryterium jest suma wartości korali w naszyjniku. Im większa suma, tym sposób powinien być późniejszy w uporządkowaniu. Jeśli zaś mamy dwa różne sposoby utworzenia naszyjnika, które mają równą sumę wartości, to porównujemy je według porządku leksykograficznego posortowanych rosnąco list numerów użytych korali[1]*.*

Dla przykładu rozważmy sytuację, w której są cztery korale warte kolejno (zgodnie z numeracją) 3, 7, 4 i 3 bajtalary. Z takich korali naszyjnik można utworzyć na 16 sposobów. Poniżej znajduje się uporządkowanie tych sposobów zgodnie z pomysłem Bajtyny.

Numer sposobu	*Wartości wybranych korali*	*Suma wartości wybranych korali*	*Numery wybranych korali*
1	*brak*	*0*	*brak*
2	*3*	*3*	*1*
3	*3*	*3*	*4*
4	*4*	*4*	*3*
5	*3 3*	*6*	*1 4*
6	*3 4*	*7*	*1 3*
7	*7*	*7*	*2*
8	*4 3*	*7*	*3 4*
9	*3 7*	*10*	*1 2*
10	*3 4 3*	*10*	*1 3 4*
11	*7 3*	*10*	*2 4*
12	*7 4*	*11*	*2 3*
13	*3 7 3*	*13*	*1 2 4*
14	*3 7 4*	*14*	*1 2 3*
15	*7 4 3*	*14*	*2 3 4*
16	*3 7 4 3*	*17*	*1 2 3 4*

Bajtyna chciałaby stworzyć naszyjnik, który ma k-ty numer w uporządkowaniu. Pomóż jej!

[1] Ciąg numerów korali $i_1, \ldots, i_p$ jest mniejszy leksykograficznie od ciągu numerów korali $j_1, \ldots, j_q$, jeśli albo pierwszy ciąg jest początkowym fragmentem drugiego (czyli $p < q$, $i_1 = j_1, \ldots, i_p = j_p$), albo na pierwszej pozycji, na której ciągi te różnią się, pierwszy ciąg ma mniejszy element niż drugi (czyli istnieje takie $u \in \{1, \ldots, \min(p, q)\}$, że $i_1 = j_1, \ldots, i_{u-1} = j_{u-1}$ oraz $i_u < j_u$).

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie dodatnie liczby całkowite n i k oddzielone pojedynczym odstępem, określające liczbę korali oraz żądany numer sposobu utworzenia naszyjnika według uporządkowania opisanego powyżej. W drugim wierszu wejścia znajduje się ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami – wartości kolejnych korali.

Możesz założyć, że Bajtyna nie pomyliła się i rzeczywiście istnieje co najmniej k różnych sposobów utworzenia jej naszyjnika.

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą, oznaczającą sumę wartości korali w znalezionym rozwiązaniu. W drugim wierszu wyjścia należy wypisać ciąg numerów korali użytych w naszyjniku w kolejności rosnącej, rozdzielając liczby pojedynczymi odstępami.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4 10`	`10`
`3 7 4 3`	`1 3 4`

Testy „ocen":

`1ocen`: *$n = 10$, wszystkie korale mają wartość 1,*

`2ocen`: *$n = 9$, wartości korali są kolejnymi potęgami dwójki,*

`3ocen`: *$n = 11$, jest jeden koral wart 1 bajtalar oraz 10 korali wartych 10^9 bajtalarów, zaś poprawne rozwiązanie używa wszystkich jedenastu korali.*

`4ocen`: *$n = 1\,000\,000$, $k = 10$, wartości kolejnych korali są kolejnymi liczbami od 1 do $1\,000\,000$.*

Ocenianie

Zestaw testów dzieli się na podane poniżej podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby grup testów.

We wszystkich podzadaniach zachodzą warunki $n, k \leqslant 1\,000\,000$ oraz $a_i \leqslant 10^9$.

Jeśli odpowiedź dla danego testu nie jest prawidłowa, jednak pierwszy wiersz wyjścia (suma wartości korali w znalezionym rozwiązaniu) jest prawidłowy, wówczas przyznaje się połowę liczby punktów za dany test (oczywiście odpowiednio przeskalowaną w przypadku przekroczenia przez program połowy limitu czasowego). Dzieje się tak, nawet jeśli drugi wiersz wyjścia jest nieprawidłowy, nie został wypisany lub gdy wypisano więcej niż dwa wiersze wyjścia.

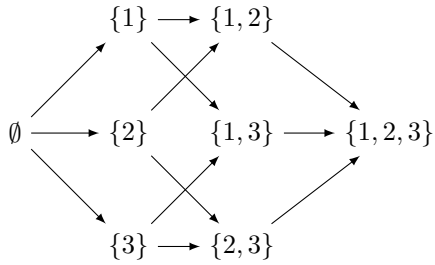
Podzadanie	Dodatkowe warunki	Liczba punktów
1	$n \leqslant 20$, $k \leqslant 500\,000$	*8*
2	$n \leqslant 60$, $k \leqslant 50\,000$	*12*
3	$n \leqslant 3\,000$, $n \cdot k \leqslant 10^6$, $a_i \leqslant 100$	*14*
4	$n \cdot k \leqslant 10^6$	*16*
5	$n \cdot k \leqslant 10^7$	*20*
6	*brak*	*30*

Rozwiązanie

Graf naszyjników

Liczba naszyjników, które można utworzyć z podanych na wejściu korali, wynosi aż 2^n, jednak zadanie dotyczy wypisania k-tego w kolejności naszyjnika. Z uwagi na stosunkowo niewielki limit na wartość liczby k (do miliona) rozsądnym podejściem wydaje się rozpatrywanie naszyjników po kolei.

Rozważmy skierowany graf, w którym wierzchołki reprezentują różne naszyjniki. Każda krawędź prowadzi do wierzchołka reprezentującego naszyjnik zawierający jeden dodatkowy koral względem naszyjnika reprezentowanego przez początek krawędzi.



Rys. 1: Graf naszyjników dla $n = 3$.

Krawędziom można przypisać wagi równe różnicy wartości naszyjników na końcach krawędzi (czyli wartości dodanego korala). Zadanie wówczas sprowadza się do znalezienia k-tego najbliższego wierzchołka od źródła (przy odpowiednim rozpatrywaniu remisów), przy czym źródłem jest wierzchołek reprezentujący pusty naszyjnik.

Algorytm Dijkstry

W tym miejscu wypada przypomnieć algorytm Dijkstry, który służy do znajdowania najkrótszych ścieżek z ustalonego źródła s do wszystkich pozostałych wierzchołków grafu $G = (V, E)$, przy założeniu nieujemnych wag krawędzi:

```
1: procedure Dijkstra(s, (V, E))
2: begin
3:     foreach v ∈ V do d[v] := ∞;
```

```
4:    d[s] := 0;
5:    C := ∅;
6:    for i := 1 to |V| do begin
7:      u := argmin{d[x] : x ∈ V \ C};
8:      foreach uv ∈ E do
9:        if d[v] > d[u] + c(u, v) then
10:         d[v] := d[u] + c(u, v);
11:     C := C ∪ {u};
12:   end
13: end
```

W każdym obiegu głównej pętli jako u przyjmuje się wierzchołek o najmniejszym oszacowaniu odległości ze źródła (najmniejszej wartości $d[u]$), który nie został jeszcze przetworzony (wierzchołki przetworzone dodawane są do zbioru C). Przetwarzanie wierzchołka sprowadza się do przeanalizowania możliwości poprawy oszacowania odległości ze źródła dla wszystkich końców krawędzi z niego wychodzących.

Poprawność algorytmu wynika z faktu, że w każdym obiegu głównej pętli oszacowanie odległości ze źródła s do przetwarzanego wierzchołka u jest w istocie już poprawnie obliczoną odległością z s do u. Dowód tego faktu można przeprowadzić indukcyjnie; patrz np. książka [6]. Dobrym ćwiczeniem na sprawdzenie zrozumienia tego dowodu jest zastanowienie się, w których miejscach dowodu korzysta się z faktu, że wagi krawędzi są nieujemne.

W standardowych implementacjach algorytmu Dijkstry zbiór $V \setminus C$ przechowywany jest na kolejce priorytetowej, implementowanej na przykład za pomocą kopca binarnego lub, w języku C++, struktury `std::priority_queue` czy struktury `std::set`. Przechowywanie zbioru w kolejce priorytetowej pozwala znajdować wierzchołek u o najmniejszym oszacowaniu odległości w czasie $O(\log |V|)$. Należy jeszcze pamiętać, że po zmianie oszacowania odległości dowolnego wierzchołka uporządkowanie zbioru nieprzetworzonych wierzchołków może się zmienić. Implementacje kolejek priorytetowych z użyciem kopca binarnego lub struktury `std::set` potrafią sobie z tym radzić z użyciem $O(\log |V|)$ porównań i zmian w pamięci, co powoduje, że algorytm ma złożoność czasową $O((|V| + |E|) \log |V|)$. Istnieją implementacje kolejek priorytetowych, które przy zmniejszeniu oszacowania odległości wierzchołka potrafią uporządkować zbiór w zamortyzowanym czasie $O(1)$[1], co poprawia złożoność algorytmu do $O(|V| \log |V| + |E|)$. Są one jednak dość skomplikowane, a w praktycznym zastosowaniu nie można zaobserwować ich asymptotycznej przewagi.

W naszym zadaniu należałoby odrobinę zmodyfikować algorytm Dijkstry, aby poprawnie rozpatrywał remisy, czyli sytuacje, w których oszacowanie odległości dla dwóch lub więcej różnych wierzchołków jest takie samo. Oczywiście należy po prostu porównać leksykograficznie naszyjniki reprezentowane przez te wierzchołki zgodnie z treścią zadania. Problemem jest jednak rozmiar grafu. Wierzchołki reprezentują bowiem naszyjniki, a tych jest (jak już wcześniej zauważyliśmy) 2^n. Grafu naszyjników nie trzeba jednak tworzyć w pamięci, a na kolejce priorytetowej wystarczy pamiętać wierzchołki, dla których została już znaleziona (niekoniecznie najkrótsza) ścieżka ze źródła, którym jest wierzchołek reprezentujący pusty naszyjnik.

[1] Chodzi między innymi o kopiec Fibonacciego ([6]).

Zastosowanie najkrótszych ścieżek do rozwiązania zadania

Siła algorytmu Dijkstry w zastosowaniu do tego zadania tkwi w tym, że przetwarza on wierzchołki grafu (naszyjniki) w kolejności takiej, której chciała Bajtyna. Algorytm zatem można przerwać po przetworzeniu k wierzchołków, a naszyjnik reprezentowany przez ów k-ty wierzchołek zwrócić na wyjście.

Łącznie liczbę operacji wykonywanych na kolejce priorytetowej będzie można oszacować przez $O(nk \log(nk))$, ponieważ przetworzenie każdego naszyjnika wprowadzi co najwyżej n zmian do kolejki priorytetowej (zmianą jest tutaj dodanie nowego naszyjnika lub poprawienie oszacowania odległości). Takie rozwiązanie powinno zaliczyć pierwsze dwa podzadania, a przy efektywnej implementacji miało szansę zaliczyć także podzadania trzecie i czwarte, co prowadziło do wyniku rzędu 20–50% punktów.

Na koniec, warto jeszcze zastanowić się nad tym, ile czasu zajmuje porównanie naszyjników. Jeśli naszyjniki mają różną wartość, a implementacja ją przechowuje, wówczas porównanie może się odbyć w czasie stałym. Jeśli jednak naszyjniki mają równą wartość, wówczas należy je porównać leksykograficznie względem numerów korali, co kosztuje czas proporcjonalny do długości krótszego z naszyjników. Teoretycznie, w najgorszym przypadku porównywane naszyjniki mają $O(n)$ korali. W naszej sytuacji możemy jednak podać dużo lepsze oszacowanie. Zauważmy, że jeśli naszyjnik ma więcej niż $\lceil \log k \rceil$ korali, to jest więcej niż $2^{\lceil \log k \rceil} - 1 \geqslant k - 1$ podzbiorów jego korali, które na pewno będą miały mniejszą wartość. A zatem k najwcześniejszych w kolejności naszyjników ma $O(\log k)$ korali i tyle czasu w najgorszym przypadku zajmuje ich porównanie.

Redukcja liczby naszyjników na kolejce priorytetowej

Powyższe rozwiązanie ma istotny problem: dla każdego przetwarzanego naszyjnika liczba zmian w kolejce priorytetowej jest rzędu $O(n)$. Wynika to stąd, że do naszyjnika, który ma c korali, kolejny koral można dołożyć na $n - c$ sposobów. Intuicja podpowiada, że tak duża liczba krawędzi grafu nie jest konieczna. Do każdego wierzchołka reprezentującego naszyjnik c-koralowy prowadzi bowiem $c!$ ścieżek, bo dla każdej permutacji korali w naszyjniku, można je dokładać w tej właśnie kolejności. Jedna ścieżka byłaby wystarczająca.

Niech $I = (i_1, i_2, \ldots, i_n)$ oznacza listę numerów korali w kolejności od najmniejszych do największych wartości (remisy rozstrzygamy dowolnie). Zmieńmy teraz definicję krawędzi w grafie naszyjników. Z wierzchołka reprezentującego naszyjnik $B = \{i_{j_1}, i_{j_2}, \ldots, i_{j_c}\}$, przy czym $j_1 < j_2 < \ldots < j_c$, będą wychodzić dwie krawędzie (o ile $j_c \neq n$):

a) do naszyjnika, w którym koral i_{j_c} zastępujemy koralem i_{j_c+1},

b) do naszyjnika z dodanym koralem i_{j_c+1}.

Tym razem źródłem przeszukiwania będzie wierzchołek reprezentujący naszyjnik $\{i_1\}$ i będziemy poszukiwali $(k-1)$-szego najbliższego wierzchołka ze źródła. Zauważmy, że teraz do wierzchołka reprezentującego naszyjnik $\{i_{j_1}, i_{j_2}, \ldots, i_{j_c}\}$ prowadzi już tylko jedna ścieżka: taka, w której najpierw koral i_1 zamieniamy sukcesywnie w koral i_{j_1},

następnie dodajemy koral i_{j_1+1}, po czym sukcesywnie zamieniamy go w i_{j_2}, itd. Trzeba będzie tylko pamiętać, aby przypadek $k = 1$ obsłużyć jako przypadek szczególny.

W nowym grafie waga krawędzi ponownie będzie równa różnicy wartości naszyjników reprezentowanych przez jej końce. Tym razem nie będzie to zawsze wartość dodanego korala, gdyż niektóre krawędzie wymieniają jeden koral na inny. Nadal jednak wszystkie krawędzie mają nieujemne wagi, bo ewentualna wymiana korala zastępuje go koralem o co najmniej takiej samej wartości.

W nowym grafie również można zastosować algorytm Dijkstry. Tym razem na kolejce priorytetowej znajdzie się tylko $O(k)$ naszyjników, gdyż przetworzenie każdego naszyjnika może dodać do kolejki co najwyżej dwa nowe. Na początku potrzebne jest posortowanie ciągu n korali według wartości. Następnie wykonujemy $O(k)$ operacji na kolejce priorytetowej, z których każda wymaga wykonania $O(\log k)$ porównań naszyjników. Jedno porównanie naszyjników kosztuje $O(\log k)$ operacji, zakładając, że korale w każdym naszyjniku pamiętamy zarówno w porządku niemalejących wartości (co służy do generowania krawędzi wychodzących), jak i w porządku numerów (co jest potrzebne właśnie do wykonywania porównań). Dodatkowo, wygenerowanie krawędzi wychodzących z danego wierzchołka zajmuje czas $O(\log k)$. Ostatecznie otrzymamy rozwiązanie działające w czasie $O(n \log n + k \log^2 k)$. Tego typu rozwiązanie otrzymywało 80–100% punktów w zależności od jakości implementacji. Przykładowy kod znajduje się w pliku `kors5.cpp`.

Szybsze generowanie naszyjników

W rozwiązaniu wzorcowym najpierw wyznaczymy listę L, w której dla każdego z k pierwszych naszyjników zapamiętamy jego wartość (v) oraz najmniejszy numer korala (y). Pokażemy dalej, że taka lista pozwoli nam odtworzyć numery wszystkich korali k-tego z kolei naszyjnika. Zysk, jaki z tego osiągniemy, będzie taki, że podczas generowania listy L nie będziemy musieli pamiętać wszystkich korali w poszczególnych naszyjnikach, dzięki czemu czas porównywania naszyjników zmniejszy się do stałego.

Przypomnijmy, że do generowania krawędzi w zmodyfikowanym grafie naszyjników wystarczy pamiętać dla danego naszyjnika numer korala o największej wadze, czyli numer najpóźniejszego korala w kolejności listy I. Oznaczmy numer tego korala w danym naszyjniku jako x. Dodatkowo, wymiana tego korala na inny (krawędź typu a)) może czasami spowodować zmianę numeru korala y. Aby sobie z tym poradzić, będziemy także pamiętać najmniejszy numer korala w naszyjniku z pominięciem tego o największej wadze (z). (Jeśli koral z nie istnieje, przyjmiemy, że $z = \infty$). Wreszcie poza numerem korala x zapamiętamy także jego pozycję p na liście I (tj. $i_p = x$). Ostatecznie dla każdego naszyjnika będziemy pamiętali tylko pięć liczb: v, x, y, z i p.

Upewnijmy się teraz, że przy przejściu krawędzią w grafie naszyjników będziemy w stanie zaktualizować przechowywane informacje. Przechodząc krawędzią typu a), koral x zastępujemy koralem $x' = i_{p+1}$ (i mamy $p' = p + 1$), co może wpłynąć na zmianę y: $y' = \min(z, x')$; natomiast z nie zmienia się ($z' = z$). Natomiast wskutek przejścia krawędzią typu b) po prostu dochodzi nam nowy koral $x' = i_{p+1}$ (i znów mamy $p' = p + 1$); to oznacza, że $z' = y$, natomiast $y' = \min(y, x')$. W obydwu przypadkach łatwo aktualizujemy wartość naszyjnika.

W kolejce priorytetowej naszyjniki porównujemy tylko według wartości v oraz, w drugiej kolejności, numeru korala y. Zauważmy, że to *nie wystarcza* do porównania naszyjników według kryterium Bajtyny, jednak pozwala w poprawny sposób wygenerować wszystkie elementy listy L jako $k-1$ pierwszych wierzchołków przetworzonych w algorytmie Dijkstry (na początek listy dodajemy pusty naszyjnik).

Przyjrzyjmy się teraz, jak za pomocą listy L możemy odzyskać wynik. Niech v_i oraz y_i oznaczają wartość i najmniejszy numer korala w i-tym naszyjniku z listy. Wiemy wówczas, że pierwszy koral k-tego naszyjnika to v_k, i możemy go wypisać. Niech m oznacza liczbę elementów listy L o wartości v_k i koralu y_k. Wówczas kolejne korale k-tego naszyjnika będą takie same jak korale m-tego naszyjnika na liście spośród tych o wartości $w = v_k - a_{y_k}$ i najmniejszym numerze korala większym niż v_k. Naszyjnik ten możemy znaleźć, cofając się na liście L do pierwszego naszyjnika o wartości w i najmniejszym koralu większym niż v_k, a następnie idąc o $m-1$ naszyjników do przodu; niech i oznacza numer tak określonego naszyjnika. Wówczas numer korala y_i tego naszyjnika to zarazem drugi w kolejności numer korala k-tego naszyjnika. Rozumowanie to powtarzamy, począwszy od naszyjnika numer i, aż do znalezienia wszystkich numerów korali k-tego naszyjnika. Całe odzyskiwanie działa w czasie $O(k)$.

Przykład 1. Oto jak wygląda lista L dla przykładu w treści zadania ($k = 10$):

i	1	2	**3**	4	5	6	7	**8**	9	**10**
v_i	0	3	**3**	4	6	7	7	**7**	10	**10**
y_i	-	1	**4**	3	1	1	2	**3**	1	**1**

Przypomnijmy, że $a_1 = 3$, $a_2 = 7$, $a_3 = 4$ i $a_4 = 3$.

Pierwszy koral szukanego, dziesiątego w kolejności naszyjnika ma numer $y_{10} = 1$. Dalej, są $m = 2$ naszyjniki na liście o wartości $v_i = v_{10} = 10$ i koralu $y_i = y_{10} = 1$. Stąd kolejny koral dziesiątego naszyjnika jest taki sam jak pierwszy koral drugiego spośród naszyjników, które mają wartość $v_i = v_{10} - a_{y_{10}} = 7$ i najmniejszy numer korala $y_i > y_{10} = 1$. Szukany koral to zatem $y_8 = 3$ z naszyjnika numer 8. Kontynuując ten proces, stwierdzamy, że jest tylko $m = 1$ naszyjnik o wartości $v_i = v_8 = 7$ i koralu $y_i = y_8 = 3$. Tak więc kolejny koral dziesiątego naszyjnika jest taki sam jak pierwszy koral jedynego naszyjnika, który ma wartość $v_i = v_8 - a_{y_8} = 3$ i najmniejszy numer korala $y_i > y_8 = 3$. Jest to zatem koral numer $y_3 = 4$ z naszyjnika numer 3.

Złożoność rozwiązania zmniejsza się do $O(n \log n + k \log k)$, gdyż każde porównanie realizowane jest w czasie stałym, a liczba operacji na kolejce priorytetowej nie zmieniła się. Implementacja znajduje się w pliku `kor.cpp` i otrzymuje oczywiście maksymalną punktację. W praktyce jednak rozwiązanie to nie jest o wiele szybsze od poprzedniego, gdyż stały czynnik ukryty w notacji asymptotycznej jest dość duży – dla danych z zadania niewiele mniejszy od czynnika logarytmicznego w poprzednim rozwiązaniu.

Rozwiązanie alternatywne

Na zadanie można spojrzeć inaczej, bez rozważania grafu naszyjników. Zamiast tego, można próbować obliczać listy k pierwszych naszyjników w kolejności wyznaczonej

przez Bajtynę zbudowanych z wykorzystaniem wyłącznie korali o numerach $i, \ldots, n$. Listy te wyznaczymy kolejno dla $i = n + 1, \ldots, 1$.

Na początku, dla pustego ciągu dostępnych korali jest tylko jeden możliwy do uzyskania naszyjnik – pusty, o wartości równej 0.

Niech $l_1, l_2, \ldots, l_{k'}$ $(k' \leqslant k)$ oznaczają kolejne naszyjniki zgodnie z uporządkowaniem Bajtyny utworzone dla korali $a_{i+1}, a_{i+2}, \ldots, a_n$. Dodamy teraz do rozważań koral o numerze i. W tym celu utworzymy drugą listę naszyjników $m_1, m_2, \ldots, m_{k'}$, w której $m_i = l_i \cup \{i\}$ (do każdego naszyjnika dodaliśmy koral i).

Teraz wystarczy złączyć listy naszyjników l oraz m w jedną, podobnie jak scala się dwie posortowane listy w jedną w algorytmie sortowania przez scalanie. Jeśli długość listy wynikowej przekroczyła k, to można bezpiecznie odrzucić jej koniec po k-tym naszyjniku, bo żaden z usuwanych naszyjników nie będzie k-ty w kolejności (skoro jest co najmniej k wcześniejszych od każdego z nich). Rozpatrywanie sufiksów ciągu korali (zamiast np. prefiksów) pomaga w utrzymywaniu poprawnej kolejności leksykograficznej ciągu korali w przypadku rozstrzygania remisów – zawsze bowiem pierwszeństwo w przypadku równej wartości naszyjników w scalanych listach będzie miał naszyjnik z listy m (z dodanym koralem i) nad naszyjnikiem z listy l (z najwcześniejszym koralem o numerze co najmniej $i + 1$).

Rozwiązanie to można zaimplementować, aby działało w czasie $O(nk)$. Powinno zaliczać wszystkie podzadania poza ostatnim i uzyskiwać około 70% punktów. Implementacja znajduje się w pliku `kors3.cpp`.

Inne rozwiązania

Rozwiązanie wykładnicze

Narzucającym się rozwiązaniem jest wygenerowanie wszystkich 2^n naszyjników. Wówczas można je posortować lub skorzystać z algorytmu selekcji[2], co prowadzi do algorytmu o złożoności $\Omega(2^n)$, który mógł zaliczyć jedynie pierwsze podzadanie.

Implementacja znajduje się w pliku `kors1.cpp`.

Rozwiązanie oparte o wydawanie reszty

Na zadanie można też spojrzeć nieco inaczej, jako na problem wydawania reszty, gdzie nominałami są wartości korali. Z użyciem algorytmu opartego o programowanie dynamiczne możliwe jest wyznaczenie dla każdej kwoty (wartości naszyjnika) liczby sposobów jej wydania (liczby różnych naszyjników o tej wartości).

Niestety, odzyskanie k-tego naszyjnika tą metodą nie jest możliwe; możliwe jest jedynie odzyskanie jego wartości.

Rozwiązanie to pozwalało uzyskać połowę punktów za trzecie podzadanie. Jest ono zaimplementowane w pliku `korb2.cpp`.

[2] Algorytm magicznych piątek lub algorytm Hoare'a [6]