Jakub Szydło

Omówienie zadań z 27. Turnieju Indywidualnego (dla początkujących)

Zaopatrzenie (A)

W zadaniu spotykamy historyjkę o zakupach i kupowaniu parkingów. Nie zadziała tutaj żadne rozwiązanie w stylu *szukam zachłannie najtańszego parkingu/najtańszcy ciastek...*. Należy do tego podejść inaczej.

Optymalne rozwiązanie może dać nam każdy z podzbiorów sklepów. Aby nie pisać 7 ifów na każdy z możliwych pozdbiorów można zaimplementować to zgrabnie pętlą:

```
for(int i = 1; i < 8; i++){
   // zapalone bity wyznaczają które ze sklepów wybieramy w danej iteracji
   // np 3 = 011 (bitowo) -- kupujemy parking do Bitla i Bajtlandu
   // 5 = 101 -- kupujemy parking do Bitronki i Bajtlandu
   // Spośród sklepów do których kupujemy parking znajdujemy najniższe ceny prodków.
}
// Pętla przejdzie po liczbach 1,2..7, ich reprezentacje bitowe to
// wszystkie niepuste podzbiory 3-elementowe:
// 001, 010, 011, 100, 101, 110, 111
```

Traktowanie liczb jako zbiorów to przydatna szerzej technika – **maski bitowe**. Przy manipulacjach tego rodzaju przydatne są różne operacje bitowe.

Przepowiednia (D)

To drugie dosyć implementacyjne zadanie. Można zobaczyć że czas trwania każdego ze znaków zodiaku rozkłada się na dwa kolejne miesiące. Danego miesiąca "występują" więc tylko dwa znaki zodiaku. Po zidentyfikowaniu miesiąca, wystarczy sprawdzić na który z dwóch znaków tego miesiąca wskazuje dzień z daty.

Dywan (G)

W zadaniu przedstawiony jest sposób konstukcji znanego fraktalu – Dywanu Siepińskiego, a naszym zadaniem jest narysować go w prostym ASCII art.

Typową cechą fraktali jest, że są "samopodobne". Nie inaczej jest z dywanem Sierpińskiego, sposób jego konstruckji wręcz narzuca rozwiązanie rekurencyje.

Łatwo można przewidzieć rozmiar dywanu D_k, nowy kolejny poziom jest trzykrotnie szerszy i wyższy od poprzedniego. Stąd D_k jest kwadratem o boku 3^k.

Możemy zainicjować sobie globalną typu `char` na wynik procedury. Procedura powinna jako argumenty mieć: W jakim miejscu zaczyna "pisać" po tablicy oraz jaki poziom fraktalu powinna narysować (ja mam jeszcze pomocniczy parametr

– dlugosc boku na danym poziomie wywołania). Rekurencja potrzebuje też warunku zatrzymania. Tutaj jest on łatwy, gdy dochodzimy do poziomu 0, rysujemy w danym miejscu # i kończymy.

Może to wyglądać następująco:

```
void recursion(int row, int col, int level, int len){
    if(level == 0){
        tab[row][col] = '#';
        return;
    }
    len /= 3;
    recursion(row, col, level-1, len);
    recursion(row+len, col, level-1, len);
    .
    .
    .
    // jest 8 takich wywołań
}
```

Żabie skoki(B)

Po przejściu przez historyjkę możemy tak zinterpretować problem: Jakie jest najmniejsze N, że liczbę K można przedstawić jako $\pm 1 \pm 2 \pm 3 ... \pm n$, gdzie za każde $\pm$ możemy niezależnie wstawić $+$ lub $-$. Oznaczmy $S(m) := 1 + 2 + ... + m = \frac{m(m+1)}{2}$.

Niech N_0 to najmniejsze n takie, że $S(n) \geq K$. Odpowiedź musi wynosić co najmniej N_0. Jeśli $R := K - S(N_0)$ jest parzyste, okazuje się że dokładnie jest to odpowiedź.

Dlaczego? Mamy, że $1 + 2 + ... + N_0 + R = K$. Możemy zmienić znak przy wartości $\frac{R}{2}$ i zachować wartośc sumy, czyli:
$1 + 2 + ... + (\frac{R}{2} - 1) - \frac{R}{2} + ... + N_0 = K$.

Co jeśli R jest nieparzyste?

- Jeśli $N_0 + 1$ jest nieparzyste, to parzyste jest $S(N_0 + 1) - K$ i dostajemy sposób na konstrukcję rozwiązania z odpowiedzią $N_0 + 1$.
- Jeśli $N_0 + 1$ jest parzyste, to nieparzyste jest $N_0 + 2$, parzyste jest więc $S(N_0 + 2) - R$, konstruujemy rozwiązanie z odpowiedzią $N_0 + 2$.

To wszystkie przypadki.

Łańcuch ozdobny (E)

W zadaniu należy znaleźć podsłowo, r o długości co najmniej 2, w którym istnieje literka, która występuje w nim co najmniej $\lfloor \frac{|r|}{2} \rfloor + 1$ razy.

Jeśli takie podsłowo nie istnieje należy to stwierdzić wypisując `NIE`. Aż 35 punktów w tym zadaniu można było dostać za rozwiązanie sprawdzające maksimum w każdym podsłowie w złożoności $O(n^2)$.

Aby dostać w tym zadaniu 100 punktów należało dokonać sprytnej obserwacji: Jeśli w s występuje podsłowo postaci XX lub XYX, wystarczy wypisać jego indeks jego początka i końca. W przeciwnym przypadku, pomiędzy dowolną parą wystąpień pewnego znaku X występują co najmniej 2 inne znaki, zatem żadne podsłowo s nie jest zdominowane, należy wypisać więc `NIE`.

Dziwna czekolada (F)

Główną trudność stanowi operacja XOR: $i \oplus j$. Możemy jednak zauważyć, że:

1. XOR jest **operacją bitową**: każda para bitów (a, b) (gdzie $a, b \in \{0, 1\}$) daje wynik $a \oplus b \in \{0, 1\}$.
2. Jeżeli $a \oplus b = 1$, to $a \neq b$. W szczególności:

$$a, b \in \{0, 1\} \quad \text{oraz} \quad a \oplus b = 1 \quad \Longrightarrow \quad (a = 1, b = 0) \text{ lub } (a = 0, b = 1).$$

Rozbijmy teraz obliczenia na poziomie **bitów**. Niech każda liczba i (w przedziale $[1, k]$) ma w zapisie binarnym bity $i_0, i_1, i_2, \ldots$, a każda liczba j — bity $j_0, j_1, j_2, \ldots$. Wówczas:

$$i \oplus j \;=\; \sum_{c=0}^{\lfloor \log k \rfloor} \big(i_c \oplus j_c\big) \cdot 2^c.$$

Innymi słowy, $i \oplus j$ to suma wag 2^c dla tych bitów c, w których $i_c \neq j_c$.

Kluczowy pomysł: pre-sumowanie wzdłuż bitów

Aby szybciej policzyć sumę po wszystkich i i j, zdefiniujmy pomocnicze tablice:

- prep$[c][1]$ — sumę wartości A_i po tych indeksach i, które mają **zapalony** (równy 1) bit c.
- prep$[c][0]$ — sumę wartości A_i po tych indeksach i, które mają **zgaszony** (równy 0) bit c.

Formalnie:

$$\text{prep}[c][1] \;=\; \sum_{\substack{i=1 \\ (1 \ll c)\,\&\, i \neq 0}}^{k} A_i, \quad \text{prep}[c][0] \;=\; \sum_{\substack{i=1 \\ (1 \ll c)\,\&\, i = 0}}^{k} A_i.$$

(Operator $\ll$ oznacza przesunięcie bitowe w lewo, natomiast & to AND bitowy).

Obliczanie wiersz po wierszu (lub „kolumna po kolumnie”)

Chcemy obliczyć

$$\sum_{i=1}^{k}\sum_{j=1}^{k} A_i \cdot B_j \cdot (i \oplus j).$$

Możemy „zamykać” sumę **dla każdego** j w postaci:

$$B_j \cdot \left(\sum_{i=1}^{k} A_i \cdot (i \oplus j)\right).$$

Przyjrzyjmy się zatem wyrażeniu $\sum_{i=1}^{k} A_i \cdot (i \oplus j)$. Rozpisując XOR bitowo:

$$(i \oplus j) = \sum_{c} \big(i_c \oplus j_c\big) \cdot 2^c.$$

Jeżeli $j_c = 0$, wówczas chcemy dodać 2^c dla wszystkich i mających bit c równy 1; natomiast jeżeli $j_c = 1$, to dodajemy 2^c dla wszystkich i mających bit c równy 0.

Zatem:

$$\sum_{i=1}^{k} A_i \cdot (i \oplus j) = \sum_{c} \underbrace{2^c}_{\text{waga bitu}} \cdot \begin{cases} \text{prep}[c][1], & \text{gdy } j_c = 0, \\ \text{prep}[c][0], & \text{gdy } j_c = 1. \end{cases}$$

Możemy to zwięźle zapisać jako:

$$\sum_{c} 2^c \cdot \text{prep}[c][\,1 - j_c\,].$$

Następnie mnożymy tę wartość przez B_j. I tak **dla każdego** j.

Złożoność

1. **Tworzenie tablic prep**$[c][0]$ **i prep**$[c][1]$:
   Przechodzimy przez wszystkie i od 1 do k, sprawdzamy kolejne bity i i dodajemy A_i do odpowiednich sum.
   – Każde i ma do $\lfloor \log_2 k \rfloor + 1$ bitów, więc budowa wszystkich prep zajmie $O(k \log k)$.

2. **Obliczanie sumy dla każdego** j:
   – Dla każdego j potrzebujemy $O(\log k)$, by odczytać kolejne bity j i zsumować $2^c \cdot \text{prep}[c][\,1 - j_c\,]$.
   – Łącznie to też $O(k \log k)$.

W sumie otrzymujemy złożoność $O(k \log k)$, co jest wykonalne dla $k \leq 100\,000$.

Upalny dzień (C)

W tym zadaniu chcemy wyznaczyć minimalny czas, który Tomek spędzi w słońcu, jeśli porusza się na płaszczyźnie z prędkością 1 i ma do dyspozycji n cieni drzew. Każdy cień to koło (okrąg) określone przez środek (x_i, y_i) i promień r_i. Dodatkowo mamy punkt startowy (x_s, y_s) oraz punkt końcowy (x_e, y_e). Ponieważ interesuje nas wyłącznie czas spędzony poza cieniami, możemy przekształcić problem w następujący sposób: jeśli Tomek potrafi przemieścić się z jednego cienia do innego (lub do/startu lub do/końca) całkowicie w cieniu, to taka podróż „kosztuje" go 0 sekund bycia w słońcu. Natomiast jeżeli na drodze musi wyjść z cienia, to warto ustalić, o ile – dokładnie jaką długość musi przebyć poza kołami.

Kluczowa obserwacja polega na tym, że jeśli rozważamy przejście między dwoma kręgami (np. dwoma drzewami lub drzewem i startem/końcem potraktowanym jako krąg o zerowym promieniu), to najlepszą strategią Tomka jest przejście trasą odcinka łączącego ich środki. Poza cieniami będzie więc odcinek, którego długość wyniesie:

$$\max\big(0,\ d - (r_1 + r_2)\big),$$

gdzie d to euklidesowa odległość między środkami obu kręgów, natomiast r_1 i r_2 to ich promienie. Jeśli dwa koła zachodzą na siebie tak bardzo, że $d \leq r_1 + r_2$, to Tomek może przejść z jednego do drugiego nie wychodząc wcale na słońce (odpowiada to „kosztowi" 0).

Na tej podstawie można zbudować graf, w którym wierzchołkami są wszystkie środki cieni oraz dodatkowo punkt startu i punkt końca (każdy z nich możemy uważać za „koło" o promieniu 0). Między każdą parą wierzchołków (czyli między każdymi dwoma środkami) dodajemy krawędź o wadze ustalonej według opisanej formuły: $\max(0, d - (r_1 + r_2))$. Dzięki temu graf w pełni odzwierciedla, jaki jest minimalny „koszt" (w sekundach bycia w słońcu) przejścia między tymi obiektami.

Po zbudowaniu takiej sieci uruchamiamy klasyczny algorytm Dijkstry, aby wyznaczyć najkrótszą ścieżkę (pod względem sumy wag) od wierzchołka reprezentującego start do wierzchołka reprezentującego koniec. Otrzymana suma wag to właśnie minimalny czas, jaki Tomek musi spędzić w słońcu. Zauważmy, że w tym podejściu nie musimy rozważać dowolnych punktów w przestrzeni – wystarczy prześledzić możliwe przejścia między centrami kręgów, ponieważ wszelkie korzystne (i bardziej skomplikowane) ruchy dałyby tę samą lub większą długość odcinków nasłonecznionych niż bezpośrednie przemieszczanie się między granicami cieni.

W efekcie otrzymujemy pełny graf na $n + 2$ wierzchołkach (z uwzględnieniem startu i końca). Choć jest on gęsty, bo zawiera $\approx O(n^2)$ krawędzi, to implementacja Dijkstry w $O(n^2 \log n)$ jest typowo wystarczająca dla rozmiarów problemu, w których takie podejście jest rozważane.