Jacek Tomasiewicz Treść zadania | **Krzysztof Kiljan** Opracowanie | **Kamil Dębowski** Program

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

Nadajniki

Bajtazar został nowym dyrektorem zabytkowej kopalni soli pod Bajtowem. Aby zwiększyć popularność tego obiektu wśród turystów, postanowił zainstalować w korytarzach kopalni bezprzewodowy Internet.

Kopalnia składa się z n komór połączonych $n - 1$ korytarzami. Z każdej komory można przejść do każdej innej, używając korytarzy. Bajtazar postanowił rozmieścić w komorach nadajniki wi-fi tak, by Internet był dostępny w każdym z korytarzy kopalni. Aby można było korzystać z Internetu w korytarzu łączącym komory a i b, musi być spełniony co najmniej jeden z poniższych warunków:

- *w komorze a lub w komorze b znajduje się nadajnik, lub*
- *w zbiorze komór, do których można dojść z komory a lub komory b, używając co najwyżej jednego korytarza, znajdują się co najmniej dwa nadajniki.*

Bajtazar zastanawia się teraz, jaka jest minimalna liczba nadajników wi-fi, które musi rozmieścić, aby można było korzystać z Internetu w każdym korytarzu. W każdej komorze można umieścić dowolną liczbę nadajników.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dodatnią liczbę całkowitą n oznaczającą liczbę komór w kopalni. Komory numerujemy liczbami od 1 do n.

Kolejne $n - 1$ wierszy opisuje korytarze w kopalni. Każdy z nich zawiera dwie liczby całkowite a i b ($1 \leqslant a, b \leqslant n$, $a \neq b$) oddzielone pojedynczym odstępem, oznaczające, że komory o numerach a i b są połączone korytarzem.

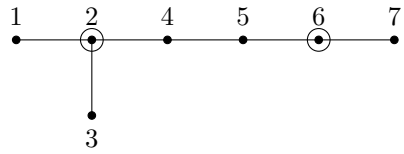
Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą minimalną liczbę nadajników, które musi rozmieścić Bajtazar.

Przykład

Dla danych wejściowych:

```
7
1 2
2 3
2 4
4 5
5 6
6 7
```

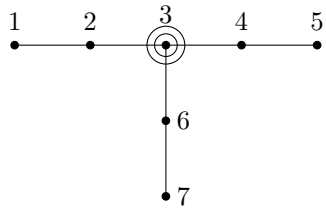


poprawnym wynikiem jest:

```
2
```

i dla danych wejściowych:

```
7
1 2
2 3
4 3
5 4
6 3
7 6
```



poprawnym wynikiem jest:

```
2
```

Wyjaśnienie do przykładów: *W pierwszym przykładzie wystarczy umieścić nadajniki w komorach o numerach 2 i 6, natomiast w drugim przykładzie wystarczy umieścić dwa nadajniki w komorze numer 3.*

Testy „ocen":

`1ocen`: $n = 16$. *Komora* i *jest połączona z komorą* $\lfloor i/2 \rfloor$ *dla* $2 \leqslant i \leqslant n$.

`2ocen`: $n = 303$. *Komora 2 jest połączona z komorami 1 oraz 3. Każda z komór 1, 2, 3 jest dodatkowo połączona z setką komór. Optymalnym rozwiązaniem jest umieszczenie dwóch nadajników w komorze 2.*

`3ocen`: $n = 200\,000$. *Komory* i *oraz* $i + 1$ *są połączone korytarzem dla* $1 \leqslant i \leqslant n - 1$.

Ocenianie

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

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 10$	*15*
2	$n \leqslant 500$	*20*
3	$n \leqslant 200\,000$, *do każdej komory prowadzą co najwyżej trzy korytarze*	*25*
4	$n \leqslant 200\,000$	*40*

Rozwiązanie

Wprowadzenie

W zadaniu mamy dany spójny graf nieskierowany o n wierzchołkach oraz $n-1$ krawędziach. Powszechnie przyjętą nazwą dla takich grafów jest *drzewo*. Drzewa charakteryzują się brakiem jakichkolwiek cykli oraz tym, że dla każdej pary wierzchołków istnieje dokładnie jedna ścieżka je łącząca.

Na potrzeby tego zadania, jako *zlewisko* krawędzi $e = (u, v)$ zdefiniujemy zbiór wszystkich wierzchołków sąsiadujących z u lub z v, co matematycznie można zapisać na przykład w następujący sposób: $Z(u, v) = \{w : (u, w) \in E \vee (v, w) \in E\}$, gdzie E to zbiór wszystkich krawędzi naszego drzewa.

W każdym z wierzchołków wstawiamy pewną nieujemną liczbę nadajników. Krawędź $e = (u, v)$ nazwiemy *spełnioną*, jeżeli zachodzi przynajmniej jeden z poniższych warunków:

- W wierzchołku u lub w wierzchołku v znajduje się co najmniej jeden nadajnik.
- W zlewisku krawędzi e znajdują się co najmniej dwa nadajniki.

W pierwszym przypadku mówimy, że krawędź jest spełniona *bezpośrednio*, a w drugim, że jest spełniona *pośrednio*.

Naszym zadaniem jest wyznaczenie najmniejszej liczby nadajników, których wstawienie gwarantuje, że wszystkie krawędzie w drzewie będą spełnione.

Rozwiązanie wykładnicze

Rozwiązywanie zadania zacznijmy od prostego spostrzeżenia.

Obserwacja 1. W żadnym wierzchołku nie warto umieścić więcej niż 2 nadajników.

Obserwacja ma oczywiste uzasadnienie – z warunków zadania wynika, że postawienie więcej niż dwóch nadajników nie może w żaden sposób wpłynąć na spełnialność którejkolwiek krawędzi. Pomysł ten pozwala nam na stworzenie pierwszego wolnego rozwiązania. W każdym z wierzchołków mamy trzy możliwości – możemy ustawić 0 nadajników, 1 albo 2.

Jako że mamy n wierzchołków, wszystkich możliwych rozstawień nadajników w wierzchołkach jest 3^n. Jeśli dane rozstawienie jest poprawne, to sprawdzamy, czy wymaga ono mniejszej liczby nadajników niż najlepsze znalezione przez nas dotychczas. Jeśli tak, to aktualizujemy potrzebną liczbę nadajników. Najprościej zaimplementować takie rozwiązanie jako funkcję rekurencyjną, którą przedstawiamy poniżej.

Przyjmujemy, że mamy napisane funkcje *calcCnt*() oraz *check*(). Pierwsza z nich ma za zadanie wyznaczyć liczbę nadajników używanych przez aktualne rozstawienie. Można ją napisać w złożoności $O(n)$, po prostu obliczając sumę wartości w tablicy *cnt*[] przechowującej liczby nadajników umieszczonych w poszczególnych wierzchołkach. Druga z funkcji sprawdza, czy dane rozstawienie nadajników powoduje spełnienie wszystkich krawędzi. Można to zrealizować, dla każdego wierzchołka zliczając

nadajniki umieszczone w jego sąsiadach (oznaczmy taką wartość przez $cntInNeigh[v]$), a potem dla każdej krawędzi (u, v) sprawdzając, czy $cnt[u] + cnt[v] \geqslant 1$ lub $cntInNeigh[u] + cntInNeigh[v] \geqslant 2$. Przy takim podejściu każde sprawdzenie zajmuje czas $O(n)$, tak więc otrzymujemy łączną złożoność czasową $O(3^n \cdot n)$.

```
 1: bestAns := ∞;
 2: function genRec(v)
 3: begin
 4:   for i := 0 to 2 do begin
 5:     cnt[v] := i;
 6:     if v = n then begin
 7:       if check() then
 8:         bestAns := min(bestAns, calcCnt());
 9:     end else genRec(v + 1);
10:   end
11: end
```

Funkcję wywołujemy jako $genRec(1)$. Rozwiązania podobne do powyższego można znaleźć w plikach `nads1.cpp` oraz `nads2.cpp`. Na zawodach poprawna implementacja takiego rozwiązania pozwalała na zdobycie między 10 a 15 punktów.

Rozwiązanie wzorcowe

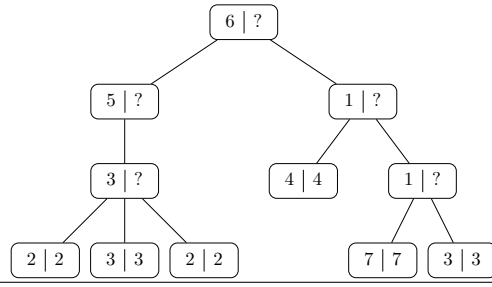
Nie jest trudno zauważyć, że przy ograniczeniach, jakie mamy dane w zadaniu ($n \leqslant 200\,000$), rozwiązania wykładnicze nie mają prawa skończyć się w żadnym sensownym czasie. Musimy więc poszukać czegoś o wiele szybszego.

Programowanie dynamiczne na drzewach

Na początku postaramy się przybliżyć, na czym polega technika programowania dynamicznego na drzewach. Jeśli Czytelnik jest z nią obeznany, może przejść do kolejnego podrozdziału, w którym będziemy próbować ją zastosować do naszego zadania. W ogólności programowanie dynamiczne polega na obliczaniu jakichś wartości dla większych egzemplarzy problemu na podstawie wyników dla mniejszych egzemplarzy. Zaczyna się od bardzo prostych przypadków, a następnie pokazuje się, jak wyniki dla mniejszych egzemplarzy złożyć w wyniki dla bardziej skomplikowanych egzemplarzy. Przykładami podstawowych problemów rozwiązywanych za pomocą programowania dynamicznego jest najdłuższy wspólny podciąg oraz (dyskretny) problem plecakowy.

Technikę tę da się również zaadaptować do wielu problemów, w których mamy do czynienia z drzewami. Wystarczy, że ukorzenimy drzewo w jednym z wierzchołków i będziemy je przechodzić od liści do korzenia. Wartość dla poddrzewa ukorzenionego w ojcu będziemy wyznaczać na podstawie wartości dla poddrzew jego dzieci. Takie podejście nazywane jest programowaniem dynamicznym *z dołu do góry*.

Dla lepszego zrozumienia, spróbujemy zobrazować to na przykładzie następującego prostego zadania: Mamy dane drzewo ukorzenione w wierzchołku numer 1. Każdy wierzchołek zawiera pewną liczbę monet v. Chcemy być w stanie w czasie $O(1)$ odpowiadać na zapytania postaci „Ile jest monet w danym poddrzewie?”.

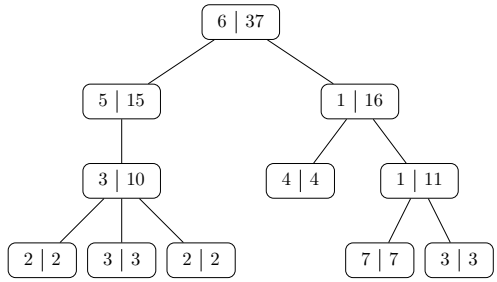


Rys. 1: Lewa wartość oznacza liczbę monet w wierzchołku, prawa – w całym poddrzewie.

W tym przypadku wnioski są oczywiste – dla każdego wierzchołka v szukana wartość jest równa jego liczbie monet $coins[v]$ powiększonej o liczbę monet w poddrzewach jego dzieci. Takie rozwiązanie możemy zrealizować za pomocą poniższego pseudokodu:

```
1:  function dfs(v)
2:  begin
3:    vis[v] := true;
4:    dp[v] := coins[v];
5:    foreach (v, child) ∈ E do
6:      if not vis[child] then begin
7:        dfs(child);
8:        merge(v, child);
9:      end
10: end
11: function merge(parent, child)
12: begin
13:   dp[parent] += dp[child];
14: end
```

Procedurę wywołujemy jako $dfs(1)$. Wartości wyznaczone za pomocą powyższego algorytmu dla drzewa z rysunku 1 można obejrzeć na rysunku 2.



Rys. 2: Drzewo z wyznaczonymi wartościami.

Stanem w programowaniu dynamicznym nazywamy egzemplarz podproblemu, dla którego obliczamy wartość. W powyższym przykładzie stan odpowiada poddrzewu oryginalnego drzewa, identyfikowanemu poprzez wierzchołek będący korzeniem poddrzewa.

Warto wspomnieć, że w programowaniu dynamicznym można wyróżnić dwa podejścia, tzw. *w tył* oraz *w przód*.

W podejściu *w tył* (częściej spotykanym) ustalamy pewien stan, którego wartość chcemy obliczyć, i wyznaczamy ją na podstawie już znanych wartości wyliczonych dla jego poprzedników, czyli stanów, od wartości których zależy jego wartość.

W podejściu *w przód* ustalamy pewien stan, którego wartość już znamy, i przeglądamy wszystkie stany, których wartości jeszcze nie znamy, a na które wpływ ma wartość owego ustalonego stanu (tzn. następników stanu).

Podejście *w tył* możemy zastosować, gdy dla stanu potrafimy wyznaczyć jego poprzedników, a podejście *w przód*, gdy dla stanu potrafimy wyznaczyć jego następników. W zależności od problemu, być może możemy zastosować oba podejścia lub tylko jedno z nich. Przykładowo, w przypadku naszego prostego zadania możliwe są oba podejścia. W powyższym pseudokodzie zaimplementowaliśmy podejście *w tył*, co wymaga założenia, że możemy efektywnie wyznaczyć wszystkie dzieci wierzchołka – a zatem należy je utrzymywać np. na liście sąsiedztwa.

Aby przećwiczyć technikę programowania dynamicznego na drzewie, można zmierzyć się na przykład z zadaniem *Farmercraft* z dnia próbnego finałów XXI Olimpiady Informatycznej [1].

Programowanie dynamiczne na drzewach w zadaniu *Nadajniki*

Do rozwiązania naszego zadania również można zastosować metodę programowania dynamicznego, jednak nie jest zupełnie oczywiste jak to zrobić. Dużym utrudnieniem może okazać się to, że nadajniki powodujące spełnienie krawędzi łączącej aktualnie przetwarzany wierzchołek z jego synem mogą znajdować się nie tylko w poddrzewie wierzchołka, ale również gdzieś wyżej. Łatwo jest też wpaść w pułapkę i wymyślić stany, których łączenie jest bardzo trudne lub wręcz niemożliwe do wykonania.

Całość zaczynamy oczywiście od ukorzenienia naszego drzewa w dowolnym wierzchołku (np. w tym o numerze 1). W rozwiązaniu wzorcowym dla każdego wierzchołka v będziemy wyliczać trójwymiarową tablicę o indeksach od 0 do 2. Dla ustalenia uwagi, będziemy ją oznaczać $t[inMe][inPar][later]$. Parametry używane w tej tablicy reprezentują odpowiednio:

- *inMe* – liczbę nadajników, które wstawiamy w wierzchołku v;
- *inPar* – liczbę nadajników, które są nam potrzebne w ojcu wierzchołka v ze względu na to, że któraś z krawędzi między wierzchołkiem v a jego synem jeszcze nie została spełniona;
- *later* – liczbę nadajników, których brakuje do pośredniego spełnienia krawędzi między wierzchołkiem v a jego ojcem, przy założeniu, że krawędź ta istnieje i nie jest spełniona bezpośrednio. Mogą być w tym celu wykorzystane nadajniki umieszczone w innych dzieciach ojca wierzchołka v lub w wierzchołku będącym dziadkiem wierzchołka v. Rozważana krawędź będzie spełniona bez względu na

parametr *later*, jeżeli w wierzchołku v lub w jego ojcu postawimy co najmniej jeden nadajnik.

Wartość $t[inMe][inPar][later]$ oznacza minimalną liczbę nadajników umieszczonych w poddrzewie wierzchołka v, które są zgodne z podanymi wartościami parametrów i spełniają wszystkie krawędzie w poddrzewach synów wierzchołka v.

Mając pewien korzeń v poddrzewa oraz wyliczone stany dla każdego z jego synów, będziemy chcieli wyliczyć jego tablicę w następujący sposób:

1. Składamy wyniki wszystkich synów v.
2. Wyliczamy tablicę dla v na podstawie scalonego wyniku dla synów, próbując umieszczenia w v każdej możliwej liczby nadajników.

Warto w tym momencie od razu zastanowić się, jakimi wartościami powinniśmy takie tablice zainicjować dla pojedynczych wierzchołków (co będzie wykorzystywane w liściach). Krawędzie od wierzchołka v do synów nie istnieją, zatem parametr *inPar* powinien być równy zeru. Jeżeli parametr *inMe* jest dodatni, to spełniamy także krawędź od wierzchołka v do jego ojca, więc w takim przypadku $later = 0$. Jeżeli jednak $inMe = 0$, to potrzebujemy dwóch nadajników, aby spełnić tę krawędź pośrednio, czyli $later = 2$. Jeżeli zatem $(inMe, inPar, later)$ przyjmuje którąś z wartości $(0, 0, 2)$, $(1, 0, 0)$, $(2, 0, 0)$, to wartość w tablicy programowania dynamicznego wynosi *inMe*. Wszystkie pozostałe stany reprezentują niedopuszczalny zbiór parametrów, zatem ustawiamy w nich bardzo dużą wartość, np. 10^9 (równoważną nieskończoności w tym zadaniu). Można dodatkowo poczynić obserwację, że jeżeli w dopuszczalnym rozstawieniu dla całego drzewa wszystkie nadajniki z konkretnego liścia przesuniemy do jego ojca, to także otrzymamy dopuszczalne rozstawienie o takim samym koszcie. Pozwala to założyć, że w liściach nie wstawiamy żadnych nadajników, co zmniejsza zbiór stanów, które musimy uwzględnić, do $(0, 0, 2)$.

Scalanie synów

Będziemy scalać dwie tablice – *t1* oraz *t2* – w jedną nową. Pierwsza z nich będzie odpowiadała wszystkim dotychczas scalonym synom, druga zaś temu, którego aktualnie będziemy dołączać. Parametry *inMe*, *inPar* oraz *later* będą otrzymywać przyrostki 1 i 2 w zależności od tablicy, z której pochodzą. Trzeba w tym momencie podkreślić, że wynikowa tablica nie jest tablicą odpowiadającą żadnemu konkretnemu poddrzewu, a zbiorowi poddrzew posiadających tego samego ojca (bez uwzględnienia samego ojca). Definicje parametrów *inMe*, *inPar* i *later* zmieniają się, ale w dość intuicyjny sposób:

- *inMe* – liczba nadajników, które wstawiliśmy w korzeniach tych poddrzew;
- *inPar* – liczba nadajników, które są nam potrzebne we wspólnym ojcu wszystkich uwzględnionych poddrzew, która zapewnia spełnienie wszystkich krawędzi pomiędzy korzeniami tych poddrzew a ich synami;
- *later* – liczba nadajników, których brakuje do spełnienia w sposób pośredni tych krawędzi między korzeniami poddrzew a ich wspólnych ojcem, które nie

są spełnione bezpośrednio. Mogą być one uzupełniane przez nadajniki z jeszcze nieuwzględnionych poddrzew, które mają tego samego ojca, lub z ojca wspólnego ojca.

Zastosujemy programowanie dynamiczne *w przód* (definicja znajduje się we wcześniejszej sekcji). Scalanie zrealizujemy, iterując po wszystkich możliwych wartościach parametrów tablic wejściowych i poprawiając, jeśli to możliwe, odpowiednie komórki tablicy wynikowej *res* (której wartości początkowo ustawiamy na nieskończoność).

Wyznaczanie parametrów *newInMe*, *newInPar* oraz *newLater* tablicy wynikowej przebiega następująco:

- *newInMe* jest równe $inMe1 + inMe2$.
- *newInPar* jest równe maksimum spośród *inPar1* oraz *inPar2*. Uzasadnienie jest jasne – oba warunki są spełnione wtedy i tylko wtedy, kiedy większy z nich jest spełniony.
- *newLater* równa się maksimum z wartości $later1 - inMe2$ i $later2 - inMe1$. Wartości te wynikają z tego, że nadajniki z korzenia sąsiedniego poddrzewa (brat aktualnego wierzchołka) mogą pomóc spełnić krawędź, której dotyczy *later*.

Jeśli wykonujemy jakieś operacje dodawania lub odejmowania na parametrach, to zawsze na koniec musimy je obciąć do przedziału $[0, 2]$ poprzez wzięcie minimów bądź maksimów z odpowiednich wartości. Zostało to zilustrowane w poniższym pseudokodzie.

```
1:  function merge(t1, t2)
2:  begin
3:    res := tablica wypełniona nieskończonościami;
4:    for inMe1 := 0 to 2 do
5:      for inPar1 := 0 to 2 do
6:        for later1 := 0 to 2 do
7:          for inMe2 := 0 to 2 do
8:            for inPar2 := 0 to 2 do
9:              for later2 := 0 to 2 do begin
10:               newInMe := min(2, inMe1 + inMe2);
11:               newInPar := max(inPar1, inPar2);
12:               newLater := max(0, max(later1 − inMe2, later2 − inMe1));
13:               res[newInMe][newInPar][newLater] :=
14:                 min(res[newInMe][newInPar][newLater],
15:                     t1[inMe1][inPar1][later1] + t2[inMe2][inPar2][later2]);
16:             end
17:   return res;
18: end
```

Obliczanie stanu ojca

Po scaleniu tablic t wszystkich poddrzew w jedną musimy ją jeszcze przełożyć na wynik dla samego korzenia v. Indeksy *newInMe*, *newInPar* oraz *newLater* będą określać wartości parametrów dla wierzchołka v. Będziemy poprawiać wartości w odpowiednich komórkach tablicy wynikowej *res*, iterując po wszystkich możliwych indeksach *inMe*, *inPar*, *later* zdefiniowanych jak w poprzedniej sekcji oraz po parametrze *newInMe*.

Zwróćmy uwagę na kilka faktów:

- Z definicji *inPar* wynika, że *newInMe* musi być co najmniej tak duże jak *inPar*.
- Jeśli $newInMe = 0$, to *newInPar* będzie równe staremu *later* – nadajniki, które wcześniej musieliśmy wstawić gdzieś wyżej, teraz musimy już wstawić w ojcu wierzchołka v.
- Jeśli $newInMe = 0$, to *newLater* będzie równe $2 - inMe$, czyli liczbie nadajników, których potrzebujemy do spełnienia krawędzi od wierzchołka v do jego ojca minus te, które otrzymaliśmy już z synów wierzchołka v.
- Jeśli *newInMe* jest większe od zera, to *newInPar* oraz *newLater* są równe zeru. Faktycznie, jeśli w wierzchołku v mamy jakiś nadajnik, to automatycznie wszystkie interesujące nas krawędzie stają się spełnione.

Po tych obserwacjach możemy już napisać pseudokod zmieniający tablicę sumy poddrzew na tablicę stanu przedstawiającego poddrzewo ukorzenione w wierzchołku v:

```
1:  function createParent(t)
2:  begin
3:    res := tablica wypełniona nieskończonościami;
4:    for newInMe := inPar to 2 do
5:      for inMe := 0 to 2 do
6:        for inPar := 0 to 2 do
7:          for later := 0 to 2 do begin
8:            if newInMe = 0 then begin
9:              newInPar := later;
10:             newLater := 2 - inMe;
11:           end else begin
12:             newInPar := 0;
13:             newLater := 0;
14:           end
15:           res[newInMe][newInPar][newLater] :=
16:             min(res[newInMe][newInPar][newLater],
17:                 newInMe + t[inMe][inPar][later]);
18:       end
19:   return res;
20: end
```

Dostosowanie funkcji *dfs* z sekcji o programowaniu dynamicznym do warunków naszego zadania pozostawiamy już jako ćwiczenie dla Czytelnika.

Wynik

Ostatnim krokiem, który musimy wykonać po przetworzeniu całego drzewa, jest odczytanie wyniku. Jak pamiętamy, parametr *inPar* oznacza, ile jeszcze potrzeba nadajników w ojcu wierzchołka v. W przypadku korzenia całego drzewa, ze względu na brak możliwości umieszczenia jakichkolwiek nadajników wyżej, musi być on równy zeru. Parametr *later* odnosił się do nadajników, które są wymagane, aby spełnić warunek dla krawędzi ponad wierzchołkiem v. Jednak dla korzenia całego drzewa takiej krawędzi nie ma, zatem wartość tego parametru w korzeniu jest dla nas nieistotna. W związku z tym, wybieramy stan o minimalnym wyniku spośród stanów $res[inMe][0][later]$, gdzie *res* jest tablicą wyników dla korzenia, a *inMe* i *later* przyjmują dowolne całkowite wartości z przedziału $[0, 2]$.

Całe rozwiązanie wykonuje liniową liczbę scaleń poddrzew i w każdym sprawdza $3^3 \cdot 3^3 = 729$ możliwości. Daje nam to asymptotycznie złożoność $O(n)$. Trzeba jednak być świadomym, że jest to rozwiązanie z wyjątkowo dużą stałą, zatem będzie znacznie wolniejsze od większości algorytmów o złożonościach liniowych dla podobnych rozmiarów danych.

Rozwiązanie opisane powyżej było wystarczająco dobre, żeby uzyskać maksymalną punktację. Jego kod można znaleźć w plikach `nad.cpp` oraz `nad2.cpp`.

Trochę szybciej

Zawodnicy, którzy chcieli przyspieszyć swój program, mogli to zrobić poprzez zmniejszenie stałego czynnika w czasie działania programu. Jedna z optymalizacji, które można zastosować, aby przyspieszyć najbardziej czasochłonny proces scalania poddrzew, jest taka, że jeżeli dla pojedynczego poddrzewa (ale już nie ich zbioru) zachodzi $inMe2 > 0$, to pozostałe parametry (*inPar2* oraz *later2*) są zerowe. Druga z optymalizacji polega na tym, że jeżeli $inPar2 > 0$, to spełniamy automatycznie wszystkie krawędzie od korzeni poddrzew do wspólnego ojca, zatem można założyć, że $later2 = 0$. Daje to rozwiązanie działające oczywiście cały czas w złożoności $O(n)$, jednak w każdym scaleniu zamiast 729 rozważamy jedynie $27 \cdot 7 = 189$ przypadków. Takie rozwiązania na zawodach nie były odróżniane od powyższego i również dostawały 100 punktów. Implementację takiego podejścia można znaleźć w pliku `nad1.cpp`.

Podzadanie 3: każda komora ma maksymalnie trzech sąsiadów

W zadaniu znalazło się podzadanie, w którym dodatkowo było nałożone ograniczenie górne na liczbę sąsiadów danego wierzchołka. W rzeczywistości oznacza ono, że jeśli dobrze ukorzenimy drzewo (w wierzchołku o maksymalnie dwóch sąsiadach), to otrzymamy drzewo, w którym każdy wierzchołek będzie miał co najwyżej dwóch synów.

Dzięki temu jesteśmy w stanie scalać wszystkie poddrzewa naraz i nie jest nam potrzebna obserwacja (być może trochę nieintuicyjna), że można je dołączać jedno po drugim.

Takie rozwiązanie przechodziło całe trzecie podzadanie. Można je obejrzeć w pliku `nadb2.cpp`.