Най-близък общ предшественик (LCA)

Основни понятия

- Дърво наричаме свързан граф без цикли.
- Кореново дърво е дърво, в което сме избрали един връх за корен.
- Ще казваме, че връх u е предшественик на връх v, ако u лежи на пътя от корена до v. В този случай ще казваме, че v е наследник на u.
- Върхът u е родител на връх v, ако u е предшественик на v и няма връх w, който е предшественик на v и наследник на u. В този случай ще казваме, че v е дете на u.
- Дълбочина на връх v е броят ръбове на пътя от корена до v.
- Най-близък общ предшественик на два върха u и v е върхът с най-голяма дълбочина, който е предшественик и на двата върха. Ще го означаваме с $\text{lca}(u, v)$.

Binary lifting

1.2.1 Намиране на k-тия предшественик на връх

Дадено ни е дърво и искаме да отговаряме на много заявки от вида "Намери k-тия предшественик на връх v, т.е. намери върхът, който е на разстояние k от v по пътя към корена".

Наивното решение е да обходим пътя от v към корена и да спрем, когато сме извървели k ребра. Това решение има сложност $O(k)$ за всяка заявка, което прави общата сложност $O(Q*N)$, където Q е броят заявки и N е броят върхове в дървото.

За по-бързо намиране на отговори ще направим предварителна обработка на дървото. Ще запазим за всеки връх v и всяка степен на двойката l кой е 2^l-тият предшественик на v. Така, когато ни се наложи да намерим k-тия предшественик на връх v, ще можем да го направим за $O(\log k)$ време, като разглеждаме двоичното представяне на k.
Това е, защото можем да представим k като сума на степени на две. Например, ако $k = 13$, то $13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$ и следователно, за да намерим 13-тия предшественик на връх v, можем да намерим 8-мия предшественик на v, след това 4-тия предшественик на този връх и накрая 1-вия предшественик на получения връх.

Как можем да направим тази предварителна обработка? Ще изчислим двумерния масив $up[l][v]$, където $up[l][v]$ е 2^l-тият предшественик на връх v. Първо, за $l = 0$, $up[0][v]$ е родителят на връх v. След това, за всяко $l > 0$, можем да използваме следното наблюдение:

$$up[l][v] = up[l-1][up[l-1][v]],$$

т.е. 2^l-тият предшественик на връх v е 2^{l-1}-тият предшественик на 2^{l-1}-тият предшественик на v, защото $2^l = 2^{l-1} + 2^{l-1}$.
До колко голямо l трябва да смятаме? Най-голямата възможна дълбочина на дървото е $N - 1$, така че $l = \lfloor \log_2 (N) \rfloor$ е достатъчно голямо.
Ако в масива $par[v]$ се съдържа родителя на връх v, то можем да направим предварителната обработка по следния начин(можем да считаме, че $par[root] = root$):

```
for(int v = 1; v <= N; v++)
{
   up[0][v] = par[v];
}

for(int l = 1; l <= logN; l++)
{
```

```
   for(int v = 1; v <= N; v++)
   {
      up[l][v] = up[l - 1][up[l - 1][v]];
   }
}
```

Важно е да не разменим последователността на вложените цикли, защото тогава ще се опитаме да използваме стойности от масива up, които все още не са изчислени.

Сега отговарянето на заявка “Намери k-тия предшественик на връх v” става по следния начин:

```
int get_kth_ancestor(int v, int k)
{
   for(int l = logN; l >= 0; l--)
   {
      if(k & (1 << l))
      {
         v = up[l][v];
      }
   }
   return v;
}
```

Нарочно вътрешният цикъл върви от по-големи степени на две към по-малки, за да сме констентни с последващите алгоритми, които ще разглеждаме, но може да се направи и в обратен ред.
Ясно е, че сложността за построяването на таблицата up е $O(N \log N)$, а сложността за отговаряне на една заявка е $O(\log k)$.

- **Задача:** CSES 1687. Company Queries I

1.2.2 Запазване на допълнителна информация за върховете/ребрата на дървото

Освен да пазим кой е 2^l-тият предшественик на връх v, можем да запазим и допълнителна информация за пътя от v до този предшественик. Например, ако дървото е претеглено, можем да запазим сумата/минимума/XOR-а или друга подходяща информация за теглата на ребрата по пътя от v до 2^l-тият предшественик на v. Това ще ни позволи да отговаряме на заявки от вида “Намери сумата на теглата на ребрата по пътя от v до k-тия предшественик на v” за $O(\log k)$ време.
Това може да се направи с допълнителен двумерен масив, който да пресмятаме подобно на масива up.

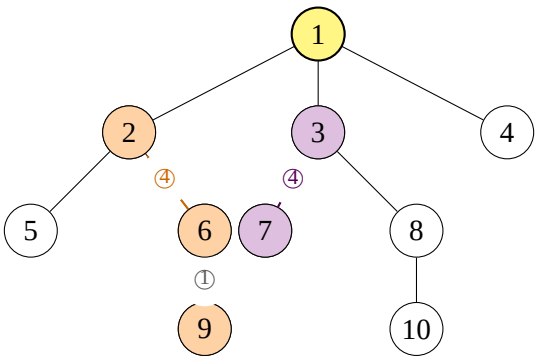
- **Задача:** Codeforces 702E. Analysis of Paths in Functional Graph

Намиране на най-близък общ предшественик на два върха

Едно от приложенията на binary lifting е намирането на най-близък общ предшественик на два върха. Нека u и v са двата върха, за които искаме да намерим lca(u, v). Можем да считаме, че u е по-дълбокият връх, т.е. $depth[u] \geq depth[v]$. Ако не е така, просто разменяме u и v.
Ще се опитаме да придвижим двата върха u и v възможно най-нагоре по дървото, допреди да достигнат своят LCA. Първо, ще придвижим u нагоре, докато достигнем същата дълбочина като v. Това можем да направим като използваме вече разгледаната функция за намиране на k-тия предшественик на връх, като в нашия случай търсим k-тия предшественик на u, където $k = depth[u] - depth[v]$.
Ако след това $u = v$, то върхът u е LCA на двата върха. Ако не, започваме да разглеждаме за всяко l от $LOGN$ до 0 дали $up[l][u] \neq up[l][v]$. Забележете, че понеже вече двата върха са на еднаква дълбочина, ако $up[l][u] \neq up[l][v]$, то $up[l][u]$ и $up[l][v]$ са различни върхове, които са предшественици съответно на u и v, но не са техен общ предшественик. Следователно, можем да придвижим u и v нагоре по дървото до техните 2^l-ти предшественици. Ако имахме $up[l][u] = up[l][v]$, то тогава щяхме да гледаме връх, който е общ предшественик на двата върха, но няма как да сме сигурни, че той е точно най-близкият общ предшественик.
Важно е да разглеждаме l от по-големи към по-малки, защото така на всяка стъпка придвижваме двата върха възможно най-нагоре, без да преминем LCA.
След това придвижване на двата върха, u и v ще са различни върхове, намиращи се непосредствено под LCA, така че отговорът на заявката ще е $up[0][u]$ (или $up[0][v]$, защото те са еднакви).
Реализацията на горния алгоритъм е следната:

```
int lca(int u, int v)
{
   if(depth[u] < depth[v]) swap(u, v);
   u = get_kth_ancestor(u, depth[u] - depth[v]);
   if(u == v) return u;
   for(int l = logN; l >= 0; l--)
   {
      if(up[l][u] != up[l][v])
      {
         u = up[l][u];
         v = up[l][v];
      }
   }
   return up[0][u];
}
```

Сложността на горния алгоритъм е $O(\log N)$.



$\text{LCA}(9,7) = 1$

① изравни dep.: $9 \to 6$

② l=3, 2: $up[l][6]$=$up[l][7]$=1 (и двете = корена) - равни, без скок

③ l=1: $up[1][6]$=$up[1][7]$=1 - равни, без скок

④ l=0: $up[0][6] \neq up[0][7]$ - скок: $6 \to 2,\ 7 \to 3$

⑤ отговор = $parent(2) = 1$

Фигура 1: Стъпките на алгоритъма за LCA върху пример с $u = 9$, $v = 7$ (дърво с $N = 10$ върха, така че $LOGN = \lfloor \log_2 10 \rfloor = 3$): първо изравняваме дълбочините ($9 \to 6$); на нива $l = 3, 2, 1$ прародителите вече съвпадат с корена и не скачаме; на ниво $l = 0$ са различни, затова скачаме едновременно ($6 \to 2$, $7 \to 3$); отговорът е $parent(2) = 1$.

- **Задача:** CSES 1688. Company Queries II

Полезни свойства на най-близкия общ предшественик

Тук ще разгледаме някои полезни свойства на най-близкия общ предшественик, които да ни служат като инструмент за решаване на задачи.

- За два върха u и v, нека $w = lca(u, v)$. Тогава разстоянието между u и v е равно на $depth[u] + depth[v] - 2 \cdot depth[w]$. Всъщност, по такъв начин могат да се изчисляват и други количества за пътя от u до v, като например сумата на теглата на ребрата по този път, ако дървото е претеглено, за което ше използваме сумата на ребрата от корена до върховете u и v и ще извадим два пъти сумата на ребрата от корена до w.
- Като следствие от горното свойство, по дадени 3 върха u, v, w, можем да проверим дали v лежи на пътя от u до w като видим дали $distance(u, v) + distance(v, w) = distance(u, w)$.
- За три върха u, v, w, нека $x = lca(u, v)$, $y = lca(v, w)$ и $z = lca(w, u)$. Върхът, който е най-близо едновременно до u, v и w, т.е. минимизира distance(u, t) + distance(v, t) + distance(w, t), е най-дълбокият от върховете x, y и z. Помислете защо!

Изчисляването на сумата от теглата на ребрата по пътя от u до v може да се направи и с binary lifting както по-горе от върха u до $lca(u, v)$ и от върха v до $lca(u, v)$. Това дори е по-добрият вариант, защото горният подход с $prefix\_sum(u) + prefix\_sum(v) - 2 * prefix\_sum(lca(u, v))$ е възможен, защото събирането има обратна операция - изваждане, но помислете как бихте направили същото, ако вместо сума трябва да намерите минимума на теглата на ребрата по пътя от u до v.

- **Задача:** Timus 1471. Distance in the Tree
- **Задача:** Codechef: Lowest Common Ancestor
- **Задача:** Codeforces 832D. Misha, Grisha and Underground
- **Задача:** Codeforces 609E. Minimum spanning tree for each edge

Sparse table

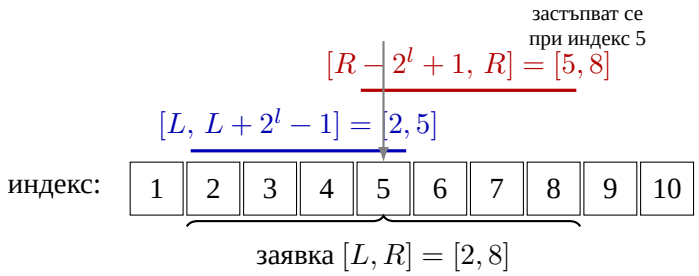
За следващия подход за намиране на най-близък общ предшественик ще ни е необходима структура от данни, която по даден масив да ни дава индекса на минимален елемент в даден интервал. Това може да бъде вършено със сложност $O(\log N)$ за всяка заявка с помощта на сегментно дърво, но в нашия случай данните, върху които работим, няма да се променят, така че можем да постигнем по-добра сложност за заявка - $O(1)$ за сметка на $O(N \log N)$ предварителна обработка. Структурата, която ще разгледаме се нарича sparse table.

Нека ни е даден масив от числа $a[1..N]$. За всеки индекс i и степен на две 2^l ще изчислим $st[l][i]$, който ще съдържа индекса на минималния елемент в интервала $[i, i+2^l-1]$. Това може да стане по следния начин:

```
for(int i = 1; i <= N; i++)
   st[0][i] = i;

for(int l = 1; l <= logN; l++)
{
   for(int i = 1; i + (1 << l) - 1 <= N; i++)
   {
      int left = st[l - 1][i];
      int right = st[l - 1][i + (1 << (l - 1))];
      if(a[left] < a[right])
         st[l][i] = left;
      else
         st[l][i] = right;
   }
}
```

Тук се възползваме от факта, че $[i, i+2^l-1]$ може да се раздели на два интервала с дължина 2^{l-1} - $[i, i+2^{l-1}-1]$ и $[i+2^{l-1}, i+2^l-1]$.
Тогава индекса на минималния елемент в интервала $[i, i+2^l-1]$ ще бъде индекса на по-малкия елемент между минималните елементи на двата интервала.

За да отговорим на заявка “Намери индекса на минималния елемент в интервала $[L, R]$”, нека разгледаме най-голямата степен на две, която е по-малка или равна на дължината на интервала $[L, R]$, т.е. $2^l \leq R-L+1$. Тогава интервалът $[L, R]$ може да се покрие от два интервала с дължина 2^l - $[L, L+2^l-1]$ и $[R-2^l+1, R]$. Наистина, ако допуснем, че има елемент, който не е в нито един от двата интервала, то тогава броят на елементите в интервала $[L, R]$ ще е поне $2^l+1+2^l = 2^{l+1}+1 > R-L+1$, което противоречи на избора на l.
Следователно индекса на минималния елемент в интервала $[L, R]$ ще бъде индекса на минималния елемент между тези два интервала.



Фигура 2: Покриване на заявка $[L, R] = [2, 8]$ (дължина 7, $2^l = 4 \leq 7$) с два интервала от дължина $2^l = 4$: $[2, 5]$ и $[5, 8]$. Те се застъпват при индекс 5 - това е допустимо, защото $min(a, a) = a$.

```
int query(int L, int R)
{
   int l = log2(R - L + 1);
   int left = st[l][L];
   int right = st[l][R - (1 << l) + 1];
   if(a[left] < a[right])
      return left;
   else
      return right;
}
```

Отбележете, че тук подинтервалите, от които вземаме минимум могат да се припокриват, но това не е проблем, защото $min(a, a) = a$. Ако имахме друга операция, която не изпълнява това свойство, тази стратегия нямаше да работи (например, ако искахме да намираме сумата на интервала вместо минимума).

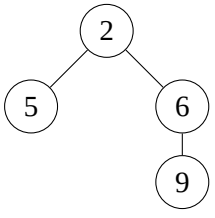
Ако не искаме да използваме функция за логаритъм, можем предварително да изчислим масив $log2[i]$, който да съдържа $\lfloor \log_2 i \rfloor$ за всяко i от 1 до N. Това може да се направи с цикъл, като се възползваме от факта, че $\lfloor \log_2 i \rfloor = \lfloor \log_2 \lfloor i/2 \rfloor \rfloor + 1$ за всяко $i > 1$.
Така изчисляването на масива ще изглежда така:

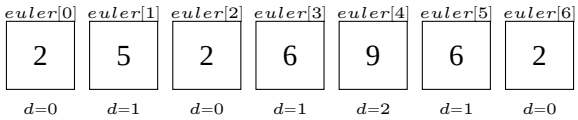
```
log2[1] = 0;
for(int i = 2; i <= N; i++)
   log2[i] = log2[i / 2] + 1;
```

Общата сложност за предварителната обработка е $O(N \log N)$, а сложността за отговаряне на заявка е $O(1)$.

Euler Tour

Друг начин да намерим LCA на два върха е като използваме обхождане в дълбочина (DFS), за да построим Euler tour на дървото. При това обхождане ние записваме всеки връх, когато го посетим за първи път, както и когато се връщаме от някое от децата му. При това обхождане ще получим последователност от върхове, която съдържа всеки връх толкова пъти, колкото е броят на децата му плюс едно (за първото посещение). Може да се види, че ако дървото има N върха, то Euler tour ще съдържа точно $2N - 1$ върха (краищата на всяко ребро се включват по веднъж, а коренът се включва и още веднъж при началото на обхождането, така $2M + 1 = 2N - 1$).





Фигура 3: Обхождане на Ойлер на поддървото с корен 2: последователността $euler = [2, 5, 2, 6, 9, 6, 2]$ с дължина $2 \cdot 4 - 1 = 7$, и съответните дълбочини $eulerDepth = [0, 1, 0, 1, 2, 1, 0]$. Тук $first[2] = 0$, $first[5] = 1$, $first[6] = 3$, $first[9] = 4$.

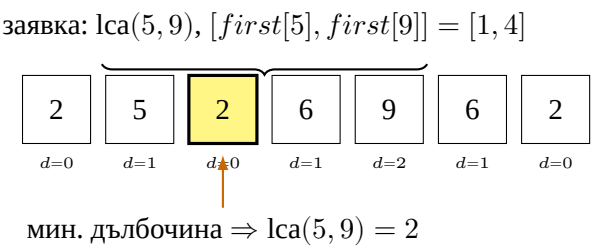
```
vector<int> euler; // stores the visited vertices during the Euler tour
int depth[MAX_N]; // depth of each vertex
int first[MAX_N]; // first occurrence of a vertex in the Euler tour

void dfs(int v, int parent, int d)
{
   first[v] = euler.size();
   euler.push_back(v);
   depth[v] = d;

   for (int to : adj[v])
   {
      if (to == parent) continue;
      dfs(to, v, d + 1);
      euler.push_back(v); // record v again after returning from a child!
   }
}
```

Важно наблюдение е, че LCA на два върха u и v е върхът с най-малка дълбочина в Euler tour между първото появяване на u и първото появяване на v. Това е така, защото ако сме добавили u в Euler tour-а, то след него и преди да се появи v, ще стоят върхове, които или са от пътя от u до v, или са наследници на върхове от този път. Следователно, върхът с най-малка дълбочина между първото появяване на u и първото появяване на v ще бъде LCA на двата върха.
Използвайки Sparse table, можем да намерим този връх с най-малка дълбочина за $O(1)$ време.

LCA с Euler Tour и Sparse Table

След като имаме Euler Tour-а и дълбочините на върховете, можем да построим Sparse table върху масива с дълбочините, но вместо да пазим минималната стойност, ще пазим индекса на върха с минимална дълбочина в интервала.

```
int lca_euler(int u, int v)
{
   int l = first[u], r = first[v];
   if (l > r) swap(l, r);
   int len = r - l + 1;
   int k = log2[len];
   int left = st[k][l];
   int right = st[k][r - (1 << k) + 1];
   if (depth[left] < depth[right])
      return left;
   else
      return right;
}
```



Фигура 4: Пример за заявка: за да намерим lca$(5, 9)$, търсим минимума на $eulerDepth$ в диапазона $[first[5], first[9]] = [1, 4]$. Минималната дълбочина ($d = 0$) е при индекс 2, който съответства на връх 2 - това наистина е lca$(5, 9)$.

Времена на влизане и излизане

Нека отново правим Euler Tour на дървото, но този път освен индекса на първото влизане на всеки връх да запазим и индекса на излизането му от обхождането.

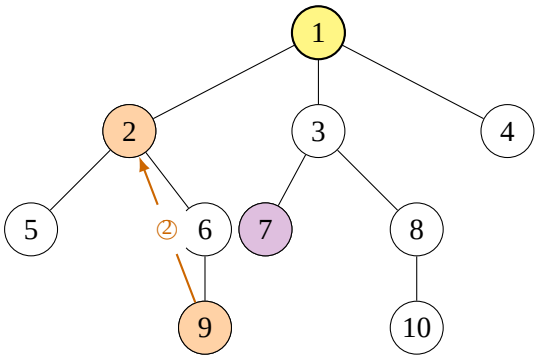
```
int in[MAX_N]; // entry index of a vertex in the Euler tour
int out[MAX_N]; // exit index of a vertex in the Euler tour

void dfs(int v, int parent, int d)
{
   in[v] = euler.size();
   euler.push_back(v);
   depth[v] = d;

   for (int to : adj[v])
   {
      if (to == parent) continue;
      dfs(to, v, d + 1);
      euler.push_back(v); // record v again after returning from a child
   }
   out[v] = euler.size() - 1; // exit index
}
```

Можем да забележим, че връх u е предшественик на връх v тогава и само тогава, когато $in[u] \leq in[v]$ и $out[v] \leq out[u]$, т.е. когато връх v е в поддървото на връх u. Използвайки това свойство, можем да реализираме трети начин за намиране на LCA, като отново използваме binary lifting:

```
bool is_ancestor(int u, int v)
{
   return in[u] <= in[v] && out[v] <= out[u];
}

int lca_in_out(int u, int v)
{
   if (is_ancestor(u, v)) return u;
   if (is_ancestor(v, u)) return v;
   for (int l = logN; l >= 0; l--)
   {
      if (!is_ancestor(up[l][u], v))
      {
         u = up[l][u];
```

```
        }
    }
    return up[0][u];
}
```

Функцията $is\_ancestor(u, v)$ проверява дали връх u е предшественик на връх v за $O(1)$ време, а функцията $lca\_in\_out(u, v)$ намира LCA на двата върха за $O(\log N)$ време. Това става по подобен начин на алгоритъма с дълбочините, но вместо да сравняваме дълбочините на върховете, опитваме да изтеглим единият от тях максимално нагоре по дървото, без да преминем LCA, като проверяваме дали върхът, до който ще се придвижим е предшественик на другия връх.



$\text{LCA}(9, 7) = 1$

① l=3, 2: $up[l][9]$=1 (корена),
`is_ancestor(1,7)` вярно ⇒ без скок

② l=1: $up[1][9]$=2,
`is_ancestor(2,7)` невярно ⇒ скок $9 \to 2$

③ l=0: $up[0][2]$=1,
`is_ancestor(1,7)` вярно ⇒ без скок

④ отговор = $up[0][2] = 1$

Фигура 5: На нива $l = 3, 2$ кандидатът вече е коренът, който е предшественик на $v = 7$, затова не скачаме. На ниво $l = 1$ кандидатът $up[1][9] = 2$ НЕ е предшественик на $v = 7$, затова скачаме ($9 \to 2$). На ниво $l = 0$ кандидатът $up[0][2] = 1$ Е предшественик на v, затова не скачаме. Отговорът е 1 - същият, както с двата предишни метода.

Euler tour и времената на влизане и излизане имат доста приложения в задачи с дървета, тъй като ни дават поглед над дървото като върху масив, върху който можем да използваме познати структури от данни като сегментни дървета. Това все пак остава извън обхвата на настоящата лекция.

Задачи

Задачите са подредени по трудност от лесни към по-трудни.

- **Задача:** CSES 1750. Planets Queries I
- **Задача:** SPOJ DISQUERY - Distance Query
- **Задача:** Codeforces 587C. Duff in the Army
- **Задача:** SPOJ QTREE2 - Query on a tree II
- **Задача:** Paths, Sofia open - autumn, 2020
- **Задача:** Codeforces 191C. Fools and Roads
- **Задача:** Turist, Летен 2019
- **Задача:** USACO New Barns

- **Задача:** Codeforces 519E. A and B and Lecture Rooms
- **Задача:** CSES 1191. Cyclic Array
- **Задача:** Codeforces 1062E. Company
- **Задача:** Codeforces 1328E. Tree Queries
- **Задача:** Codeforces 1843F2. Omsk Metro (hard version)
- **Задача:** Codeforces 828F. Best Edge Weight
- **Задача:** Codechef PSHTTR - Pishty and Tree
- **Задача:** DevSkill Motku (archived)

Автор: Даниел Трополинов