Grafuri. Parcurgerea în adâncime (DFS)

Prescurtarea provine de la denumirea algoritmului în limba engleză (*depth first search*). Alături de parcurgerea în lățime (*BFS - breadth first search*) reprezintă alt algoritm popular și util de traversare a nodurilor unui graf.

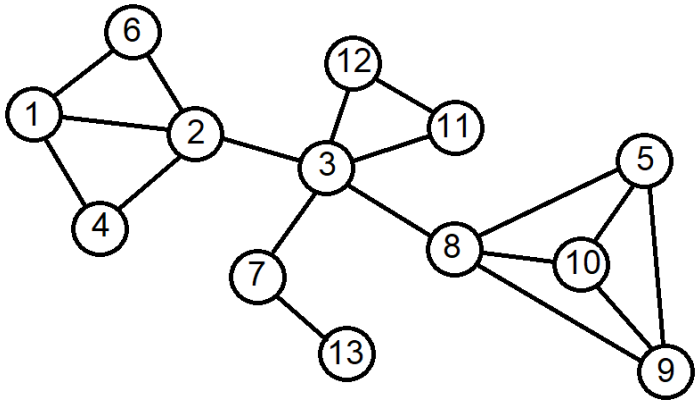
Pentru o parcurgere DFS trebuie specificat nodul de pornire. Ca și BFS, o parcurgere DFS dintr-un nod dat asigură vizitatea nodurilor din componenta conexă a acelui nod. În schimb, spre deosebire de DFS nu permite să obținem și drumul minim la nodurile în care ajungem.

Complexitatea în timp și memorie este aceeași (ca și la BFS), de ordin `n+m` (notăm cu `n` numărul de noduri și cu `m` numărul de muchii) și se obține dacă reprezentăm graful cu liste de vecini.

Are însă alte mari atuuri:
- Se codează mai ușor și mai repede;
- Permite rezolvarea unei mari categorii de probleme de programare dinamică;
- Este folosită la o serie de algoritmi dintre care enumerăm: cei bazați pe aranjarea topologică a nodurilor, cei bazați pe principiul stivei: componente biconexe, componente tare conexe, ciclu eulerian etc.

Descrierea algoritmului

Vom folosi ca exemplu următorul graf neorientat:



Graful a fost ales conex și în acest fel o parcurgere dintr-un nod oarecare al său va vizita toate celelalte noduri.
O parcurgere este o modalitate de a vizita nodurile accesibile direct sau indirect din unul stabilit de pornire, respectând regulile:
(1) La început stabilim ca fiind vizitat nodul de pornire.
(2) La un moment dat alegem un nod vizitat și marcăm ca vizitat un alt nod, încă nevizitat, care este legat de cel ales printr-o muchie.
(3) Algoritmul se încheie când nu mai poate fi aplicat pasul (`2`), adică nu ne mai putem deplasa pe muchii între un nod deja vizitat și unul încă nevizitat.

Tocmai modul în care alegem să facem următoarea vizitare (adică modul de a realiza pasul (2) la un moment dat) dă tipul de parcurgere.

La parcurgerea în lățime am arătat că mai întâi vizităm imediat vecinii nodului de pornire. Apoi trecem imediat la acești vecini și pentru fiecare dintre ei vizităm imediat toți vecinii lor încă nevizitați, și așa mai departe. Acest algoritm permite să obținem la fiecare nod un lanț cu număr minim de muchii până la nodul de pornire (lanț constituit de muchii prin care s-a avansat).

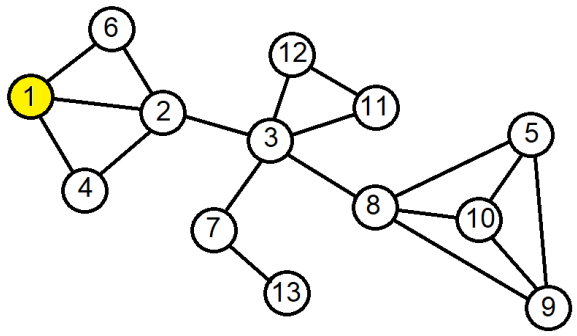
La parcurgerea în adâncime procedăm altfel. Trecem de la modul lacom (acela de a vizita imediat toți vecinii nevizitați ai unui nod, atunci când îi vine rândul) la un mod mai puțin “agresiv” cu vecinii nodului curent în sensul următor: în loc să îi vizităm imediat toți vecinii, îl vizităm doar pe primul încă nevizitat și apoi reluăm procedeul din acesta, în mod recursiv. Dacă la revenirea din recursivitate încă mai sunt vecini nevizitați ai nodului curent, abia atunci trecem la aceștia (vizităm pe următorul și apoi reluăm recursiv din el același procedeu).

Dacă am trece direct la scrierea algoritmului am vedea că sursa este mult mai scurtă decât la BFS și totodată ceea ce obținem este destul de intuitiv. Însă scopul nostru este să explicăm, pentru a înțelege cât mai bine ce se întâmplă și astfel să fim pregătiți să scriem algoritmul, mai ales că recursivitatea cu mai multe autoapeluri (care stă la baza DFS) necestită mai multă atenție la început pentru a înțelege bine lucrurile.

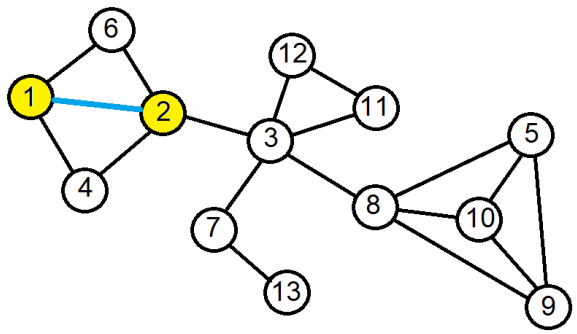
Vom exemplifica pas cu pas pe graful de mai sus. Folosim în descriere următoarele:
- considerăm `1` nodul de pornire;
- analizăm vecinii oricărui nod în ordine crescătoare a etichetei;
- colorăm cu galben nodurile vizitate;
- de câte ori avansăm dintr-un nod vizitat în unul nevizitat colorăm cu albastru muchia dintre ele.

Evident, vom face totul pe strategia de parcurgere în adâncime descrisă anterior.
Alături de explicații despre ce se întâmplă, vom afișa mereu lista nodurilor vizitate în ordinea în care acestea s-au accesat.

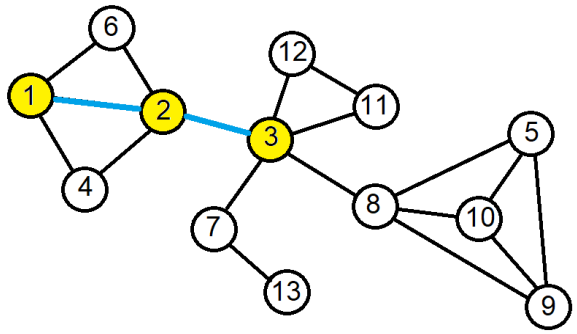
Așadar inițial avem vizitat nodul `1`.

Graful	
Nodurile vizitate	`1`

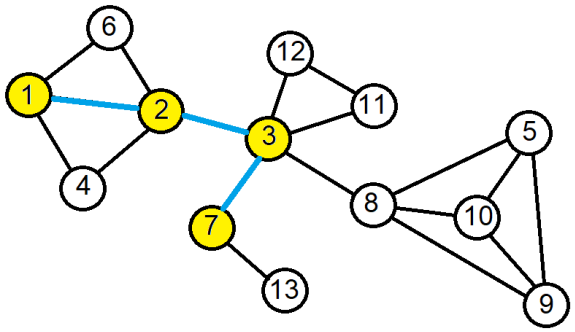
Fiind în nodul `1` mergem în primul vecin nevizitat al său. Acesta este `2`. Totodată desenăm cu albastru muchia (`1`,`2`). Vom vedea la final că folosind muchiile evidențiate se va obține un arbore de acoperire a grafului dat (un graf parțial al său care este arbore).

Graful	
Nodurile vizitate	`1`, `2`

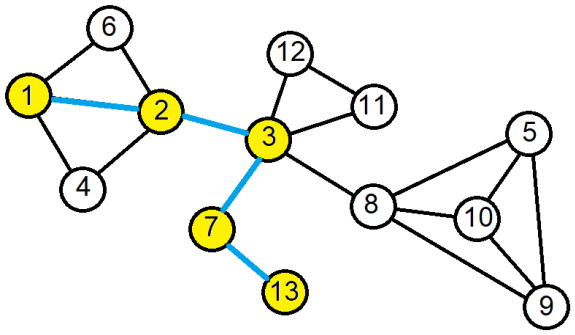
Așa cum am descris, acum procedeul se reia recursiv din noul nod, adică `2` (spre deosebire de BFS care din `1` ar fi vizitat imediat și pe `4` și pe `6`, odată cu `2`). Primul vecin nevizitat al lui `2` este `3`. Așa că vizităm pe `3`, marcăm muchia (`2`,`3`) și reluăm recursiv din `3`.

Graful	
Nodurile vizitate	`1`, `2`, `3`

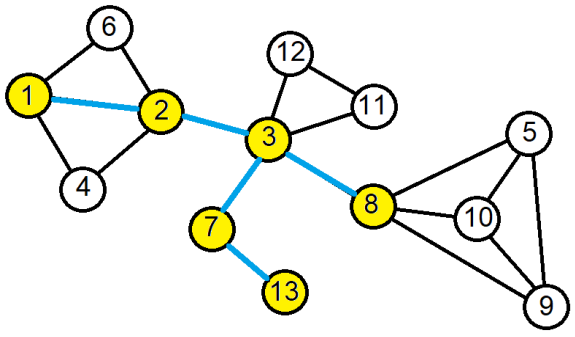
Primul vecin nevizitat al lui `3` este `7` (reamintim că am convenit că vecinii nodurilor vor fi analizați în ordinea crescătoare a etichetei). Vom vizita pe `7` și vom adăuga la arborele de acoperire muchia (`3`,`7`)

Graful	
Nodurile vizitate	`1, 2, 3, 7`

Suntem acum în `7` și singurul vecin nevizitat al său este `13`. Așadar vizităm pe `13` și evidențiem muchia (`7,13`)

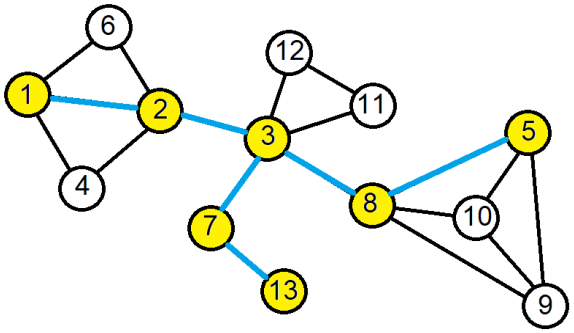
Graful	
Nodurile vizitate	`1, 2, 3, 7, 13`

Nodul `13` nu are vecini nevizitați și apelul recursiv în el se va termina fără să se mai facă altul. Astfel vom reveni în locul din care am făcut apelul recursiv pentru `13`, adică în `7`. Dacă acest nod ar mai avea vecini nevizitați acum ar fi momentul să avansăm în primul dintre ei. Dar nici `7` nu mai are vecini vevizitați, astfel și apelul recursiv pentru el se va termina, revenindu-se în apelul din care s-a mers în `7`. Revenim deci în apelul nodului `3` și continuăm să căutăm vecini nevizitați ai acestuia. Găsim și vom merge cu autoapel în `8`. Așadar marcăm modul `8` și considerăm muchia (`3,8`).

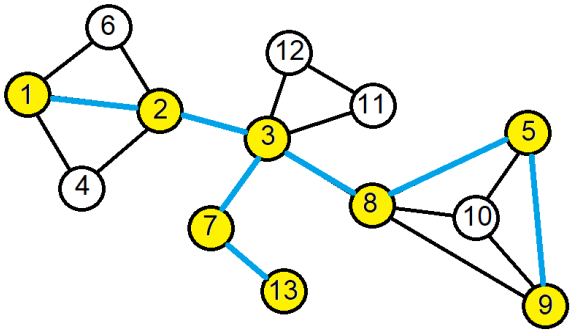
Graful	

Nodurile vizitate	1, 2, 3, 7, 13, 8

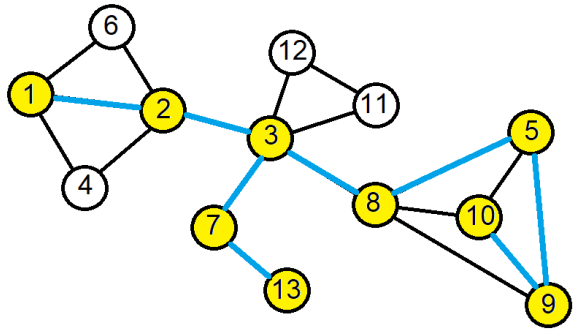
Nodul curent, 8 are un prim vecin nevizitat nodul 5. Deci mergem în 5 și adăugăm la arbore muchia (8,5)

Graful	
Nodurile vizitate	1, 2, 3, 7, 13, 8, 5

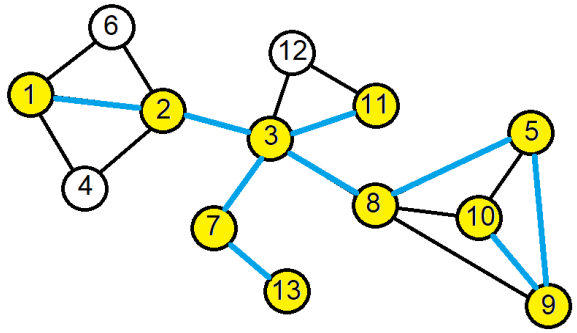
Dacă deja ați înțeles mecanismul puteți sări următorii pași. Totuși ceva răbdare să terminați parcurgerea vă va răsplăti sigur cu aprofundarea modului în care funcționează algoritmul. Așa că vom merge din 5 cu autoapel în 9, punând la arbore muchia (5,9).

Graful	
Nodurile vizitate	1, 2, 3, 7, 13, 8, 5, 9

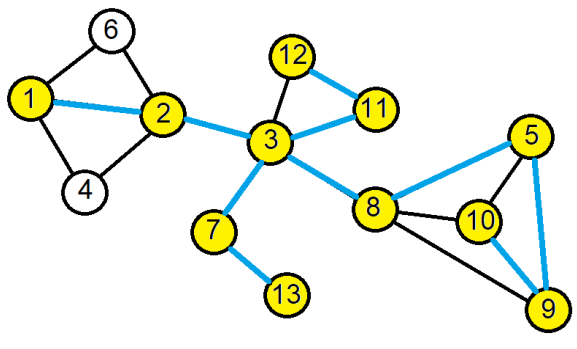
Acum din 9 se autoapelează în singurul vecin nevizitat, 10. Adăugăm deci la arborele de acoperire muchia (9,10).

Graful	
Nodurile vizitate	`1, 2, 3, 7, 13, 8, 5, 9, 10`

Urmează câțiva pași de întoarcere astfel: `10` nu mai are vecini nevizitați, deci se revine în apelul lui `9`. Nici acesta nu mai are vecini nevizitați și se va reveni în apelul lui `5`, cel din care s-a declanșat apelul lui `9`. Dar nici `5` nu mai are vecini nevizitați, și nici `8`, cel în care se revine din `5`. În fine, la încheierea apelului în `8` se revine la `3` care are în `11` și în `12` vecini nevizitați. Așadar se va merge recursiv din `3` în `11`. Vom vedea mai departe, cum cred că s-a înțeles deja, că nodul `12` nu va fi vizitat din `3` întrucât anterior se va merge în el din `11`. Așadar marcăm acum pe `11` și punem la arbore muchiea (`3`,`11`).

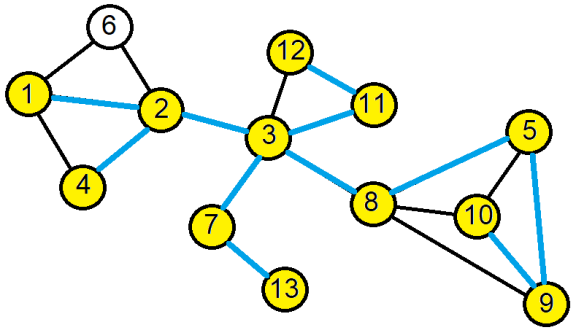
Graful	
Nodurile vizitate	`1, 2, 3, 7, 13, 8, 5, 9, 10, 11`

Cum am descris și anterior, din `11` vom merge în `12`. Marcăm astfel nodul `12` și muchia (`11`,`12`).

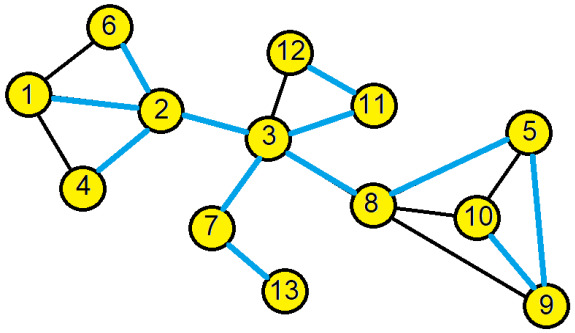
Graful	

Nodurile vizitate	1, 2, 3, 7, 13, 8, 5, 9, 10, 11, 12

Acum urmează alte întoarceri. Nodul 12 nu mai are vecini și se revine în 11, nici acesta nu mai are vecini și se revine în 3 și similar în 2. Pentru nodul 2 mai există acum vecini nevizitați așa că se va face apel recursiv în 4, considerând totodată la arbore muchia (2,4).

Graful	
Nodurile vizitate	1, 2, 3, 7, 13, 8, 5, 9, 10, 11, 12, 4

Acum nu mai avem vecini nevizitați pentru 4 așa că i se va încheia apelul și se revine în 2. Din 2 găsim ca vecin bun pe 6 în care vom face un ultim autoapel, adăugând la arbore muchia (2,6).

Graful	
Nodurile vizitate	1, 2, 3, 7, 13, 8, 5, 9, 10, 11, 12, 4, 6

Se termină deci autoapelul din 6 (acesta nemaiavând vecini nevizitați), se revine în 2, nici el nu mai are vecini nevizitați, deci revenim în locul de unde s-a făcut apelul pentru 2, adică în 1. Nici din 1 nu mai sunt vecini nevizitați așa că se va termina și acest apel. Fiind acum vorba de apelul din nodul de pornire, cel care le-a declanșat pe toate celelalte, se va încheia parcurgerea.

După ce am descris detaliat modul în care traversarea în adâncime “se plimbă” prin graf, să vedem cum arată algoritmul.

Presupunem că graful este neorientat, cu `n` noduri și cu tabloul `L` în care fiecare componentă este un vector stl ce conține lista de vecini.

```
/// declararea datelor importante
vector<int> L[MAX_N];
int viz[MAX_N];
fin>>n;
```

```
/// citirea
fin>>n>>m;;
for (i=1;i<=m;i++) {
    fin>>x>>y;
    L[x].push_back(y);
    L[y].push_back(x);
}
```

Funcția recursivă ce realizează parcurgerea se scrie astfel:

```
void dfs(int nod) {
   viz[nod] = 1;
   for (int i=0;i<L[nod].size();i++) {
      int vecin = L[nod][i];
      if (viz[vecin] == 0)
         dfs(vecin);
   }
}
```

Apoi, apelăm în main simplu: **`dfs(1);`** Noi am considerat `1` ca fiind nodul de pornire.

Detalii de implementare:

- Prima grijă când intrăm în apelul recursiv pentru un nod este să marcăm nodul ca fiind vizitat.
- Am ales varianta de a scrie repetiția folosind un indice întreg față de cea cu iterator pentru a pune mai bine în evidență variabila `i`, care trebuie să fie **locală**. Numai cu ea locală putem ca pentru nodul curent să continuăm căutarea altui vecin nevizitat la revenirea din autoapelul pentru un alt vecin al său. Astfel fiecare autoapel își păstrează propriul său contor cu evidența vecinului la care a rămas cu parcurgerea.
- Odată vizitat un nod observăm că nu mai putem face autoapel în el. Totodată apelul recursiv al fiecărui nod parcurge lista sa de vecini. În total forurile interioare vor face număr de pași egal cu `2*m` (orice încercare de a accesa un vecin este de fapt o muchie

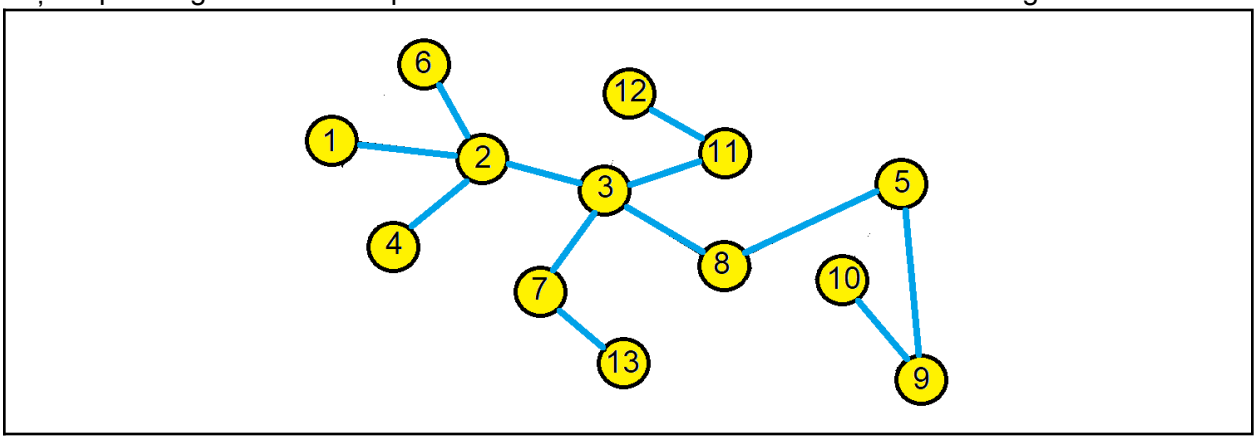
și orice muchie este vizitată dinspre ambele extremități). Astfel obținem complexitatea în timp de ordin `n+m`, la fel ca la BFS.

Mai departe vom face diverse adaptări ale algoritmului dfs prezentat și vom observa cum obținem multe alte lucruri de bază. Vom considera de fiecare dată că nodul de pornire este `1`. Nu restrângem din generalitate, dar astfel ne putem concentra pe lucrurile specifice.

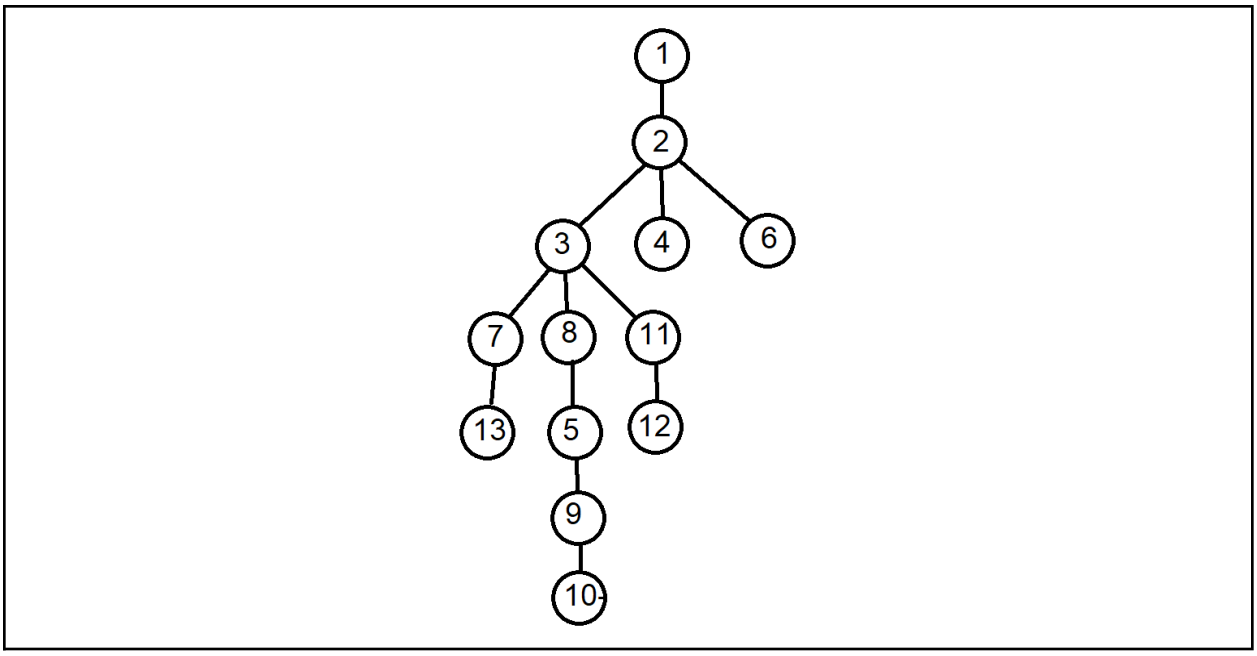
1. Folosim un parametru, notat `niv`, facem apelul inițial cu `niv = 1` și autoapelurile cu `niv+1`.

```
void dfs(int nod, int niv) {
    viz[nod] = 1;
    for (int i=0;i<L[nod].size();i++) {
        int vecin = L[nod][i];
        if (viz[vecin] == 0)
            dfs(vecin, niv+1);
    }
}

Apel inițial dfs(1, 1);
```

Pentru a descrie mai clar ce se întâmplă, să considerăm arborele de acoperire pe care l-am obținut pentru graful din exemplul de mai sus. Eliminând michiile rămase cu negrul el ar atăta:



Îl vom redesena mai jos (păstrăm aceleași noduri și muchii)



Parametrul `niv` va avea valoarea `1` la apelul în nodul inițial, `1` dar la autoapelul în `2`, pentru valoarea `niv` de la acest nivel se va transmite cu `1` mai mult, adică `2`. La fel se întâmplă cu autoapelul din `2` în `3`, deci când se intră în apelul pentru nodul `3`, valoarea `niv` pentru acest nivel va fi `3`. Trebuie remarcat că și pentru autoapelurile în `4` și în `6` se va intra tot cu `3` pentru valoarea parametrului `niv`. Acest lucru se întâmplă pentru că `niv` se comportă ca o variabilă locală, adică dacă se face autoapel, la revenirea din acesta se revine la valoarea lui `niv` pentru apelul curent (noi nu mărim pe `niv` cu `1` ci doar formăm cu el expresia `niv+1` pe care o transmitem ca dată de inițializare pentru variabila `niv` a apelului în care tocmai intrăm).
Revenind, atunci când pornim către `3`, valoarea lui `niv` de la apelul în `2` rămâne `2`, și la fel se întâmplă la revenirea din apelul către `3`, apoi la trecedea din `2` la apelul către `4` se va transmite iarăși `niv+1`, adică tot `3`.
Exemplul de mai sus arată cum putem ierarhiza pe nivele un arbore. Pe de o parte, dacă avem la intrare un graf oarecare, arborele de acoperire îl putem privi ca pe unul cu rădăcina în nodul de pornire și fiecare muchie a sa o vedem ca o trecere pe un nivel în jos. Pe de altă parte, dacă la intrare avem deja un arbore oarecate, în urma unei parcurgeri dfs îl putem privi ca pe un arbore cu rădăcină, pe nivele, în care nivelul fiecărui nod este valoarea cu care ajunge variabila niv la apelul în acel nod.
Pentru a pune mai bine în evidență arborele cu rădăcină, dar și reprezentarea sa pe nivele, vom face câteva adăugări la algoritmul dfs de mai sus.

2. Putem stoca într-un vector global nivelul fiecărui nod, dispunând de aceste valori și după încheierea parcurgerii.

```
void dfs(int nod, int niv) {
    viz[nod] = 1;
    Niv[nod] = niv;
    for (int i=0;i<L[nod].size();i++) {
```

```
        int vecin = L[nod][i];
        if (viz[vecin] == 0)
            dfs(vecin, niv+1);
    }
}
```

Apel inițial `dfs(1, 1);`

3. Pentru arborele de acoperire obținut (pe care îl privim, cum spuneam, ca pe unul cu rădăcina în nodul de pornire) putem construi un vector de părinți (sau de "tați", cum se mai spune).

O primă modalitate este ca la apelul în vecin să notăm `T[vecin] = nod;`

```
void dfs(int nod) {
    viz[nod] = 1;
    for (int i=0;i<L[nod].size();i++) {
        int vecin = L[nod][i];
        if (viz[vecin] == 0) {
            T[vecin] = nod;
            dfs(vecin);
        }
    }
}
```

Apel inițial `dfs(1);`

Se obține astfel `T` ca fiind un vector de tați global și îl putem folosi după terminarea parcurgerii (evident că în `T[1]` va rămâne `0`, în rest cele `n-1` muchii ale arborelui de acoperire sunt perechile (`i`, `T[i]`)).

O altă modalitate de a construi un vector de tați, sau de a ne referi mai direct la tatăl nodului curent, este să punem un parametru în plus la funcție, care să reprezinte tatăl lui `nod`.

```
void dfs(int nod, int tata) {
    viz[nod] = 1;
   T[nod] = tata; /// această linie e necesara doar dacă
                  /// avem nevoie de tata la finalul parcurgerii
    for (int i=0;i<L[nod].size();i++) {
        int vecin = L[nod][i];
        if (viz[vecin] == 0) {
            dfs(vecin, nod);
        }
    }
}
```

Apel inițial `dfs(1, 0);`

Dacă perechea (`nod`, `tata`) o procesăm în funcția recursivă nu mai este necesar să stocăm date și în vectorul global `T`.

4. O altă modificare de a folosi algoritmul în mod util este cea prin care păstrăm într-un vector separat parametrii din stiva de autoapeluri (vom exemplifica la problema *cerere* de la finalul materialului).

```
void dfs(int nod, int niv) {
    viz[nod] = 1;
    st[niv] = nod; /// la un moment dat in st este lantul de la
radacina la nodul curent
    for (int i=0;i<L[nod].size();i++) {
        int vecin = L[nod][i];
        if (viz[vecin] == 0)
            dfs(vecin, niv+1);
    }
}
```

Apel inițial `dfs(1, 1);`

Vectorul global `st` va reține nodurile din autoapelurile făcute dar încă neterminate. Noi când am explicat pas cu pas pe desen, spuneam des că terminăm de analizat vecinii unui nod, terminăm deci și apelul în el și revenim la nodul din care am făcut acest apel. Acesta este chiar nodul care se află imediat mai jos în stivă. Se vede clar că modul în care valorile lui `niv` cresc cu `1` și apoi revin la valoarea anterioară la terminarea autoapelurilor, are exact comportamentul unei variabile ce indică spre vârful unei stive.
Conținutul vectorului `st` în apelul curent este de fapt conținutul lanțului de la rădăcină la nodul curent în arborele de acoperire.

5. La acest subpunct presupunem că facem parcurgerea dfs pe un arbore oarecare. Evident că parcurgerea îl va "transforma" în unul cu rădăcină. Marcăm în algoritmul de mai jos ce semnificație are fiecare loc în care am putea scrie cod.

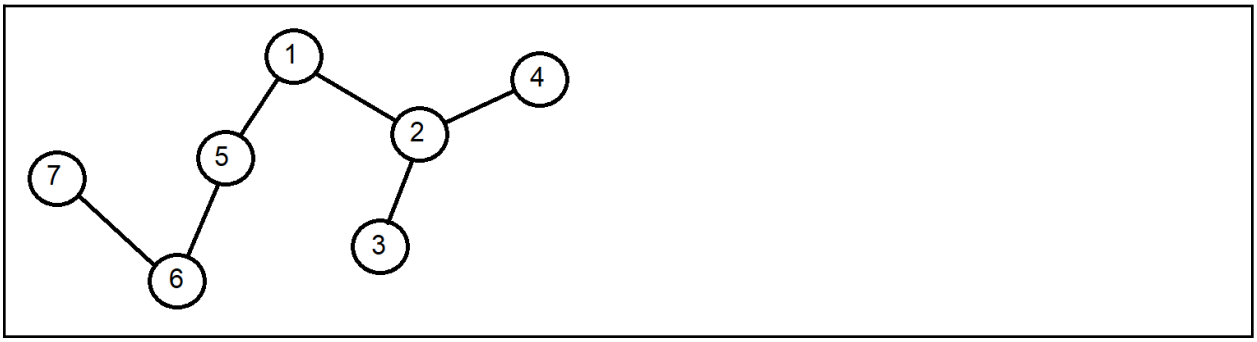
```
void dfs(int nod) {
    // tocmai intrăm în nod coborând din tatăl său
    viz[nod] = 1;
    for (int i=0;i<L[nod].size();i++) {
        int vecin = L[nod][i];
        if (viz[vecin] == 0) {
            // înainte de a coborî din nod în fiul său
            // numit aici "vecin"
            dfs(vecin);
            // când revenim în nod urcând din fiul său numit "vecin"
            // acesta este un loc foare util
            // de speculat în problemele de
            // programare dinamică întrucât putem
            // considera că avem calculată
```

```
            /// informația din fiu și o putem folosi pentru nod
        }
    }
    // tocmai am terminat de procesat toți fiii lui nod
    // și revenim în nod;
    // și acesta este un loc unde folosim la o dinamică
    // informațiile despre toți fii, existente acum în formă finală
}

Apel inițial dfs(1);
```

Probleme pentru fixarea ideilor de bază

1. **sediu**.https://www.infoarena.ro/problema/sediu

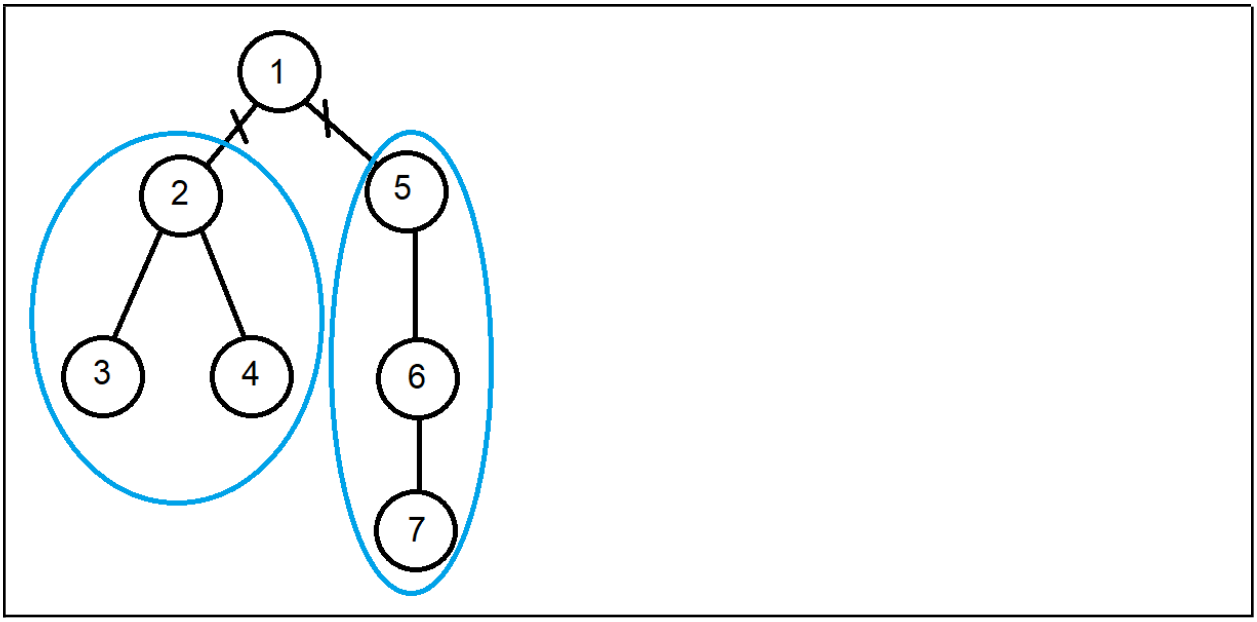
Enunțul pe scurt
Se dă un arbore oarecare din care putem elimina exact un nod. În urma acestei eliminări arborele se poate împărți în mai mulți subarbori. Ne interesează să minimizăm numărul de noduri din subarborele cu numărul maxim de noduri dintre cei rămași. Apoi ne mai interesează pentru câte noduri din arborele inițial eliminarea lor ar oferi optimul descris mai sus.



De exemplu, pentru graful de mai sus, dacă am elimina nodul `5` am obține doi subarbori: unul cu două noduri (`6` și `7`) și unul cu `4` noduri (`1`, `2`, `3`, `4`). Așadar costul eliminării lui `5` este egal cu `4` (numărul cel mai mare de noduri din unul dintre arborii rămași).
Dacă am elimina pe `2` se obțin trei subarbori (cel format doar din `3`, cel format doar din `4` și cel format din `1`, `5`, `6`, `7`). Costul operației este tot `4`.
Dacă însă alegem să eliminăm nodul `1` vom obține doi subarbori, ambii cu câte `3` noduri, deci costul operației este `3`. Nu se poate obține mai bine.

Prima soluție este de a încerca eliminarea pe rând pentru fiecare nod. Cu un nod ales pentru eliminare putem să determinăm componentele conexe ale grafului rămas și să calculăm numărul de noduri din fiecare. Determinarea componentelor conexe se face prin algoritmi de parcurgere. În general complexitatea parcurgerilor este de ordin `n+m`, dar în cazul arborilor `m = n-1`. Astfel am avea complexitate de ordin `n`. Însă noi avem de făcut de `n` ori determinarea componentelor conexe (considerând fiecare nod pentru eliminare). Obținem astfel un algoritm cu timp de calcul de ordin n^2.

O altă soluție, cu aceeași complexitate, folosește o idee ce ne va fi utilă apoi și la soluția optimă, și la alte soluții. Facem o parcurgere din nodul pe care dorim să îl eliminăm.
Dacă ștergem muchiile dintre el și fii săi este exact ce ne trebuie: el se izolează și fiecare fiu devine rădăcina unui subarbore conex pentru care ne interesează numărul de noduri.



De exemplu, la un dfs din `1`, subarborii conecși care ne interesează sunt cei cuprinși în desen în elipsele albastre.
Acum este momentul să prezentăm cel mai simplu algoritm de dinamică de arbore, un fel de sume parțiale pe arbore. *Pentru fiecare nod vom determina numărul de noduri din subarborele cu rădăcina în el.*
Pentru frunze această valoare va fi `1`. Dacă pentru nodul curent calculăm valoarea **la revenirea din autoapeluri**, avem deja informația din fiu deci o putem colecta la informația pentru `nod`. Nu uităm că trebuie să adunăm și valoarea `1` (se numără și nodul însuși).

```
void dfs(int nod) {
    viz[nod] = 1;
    S[nod] = 1;
    for (int i=0;i<L[nod].size();i++) {
        int vecin = L[nod][i];
        if (viz[vecin] == 0) {
            dfs(vecin);
            S[nod] += S[vecin];
        }
    }
}
```

Apel inițial `dfs(1);`

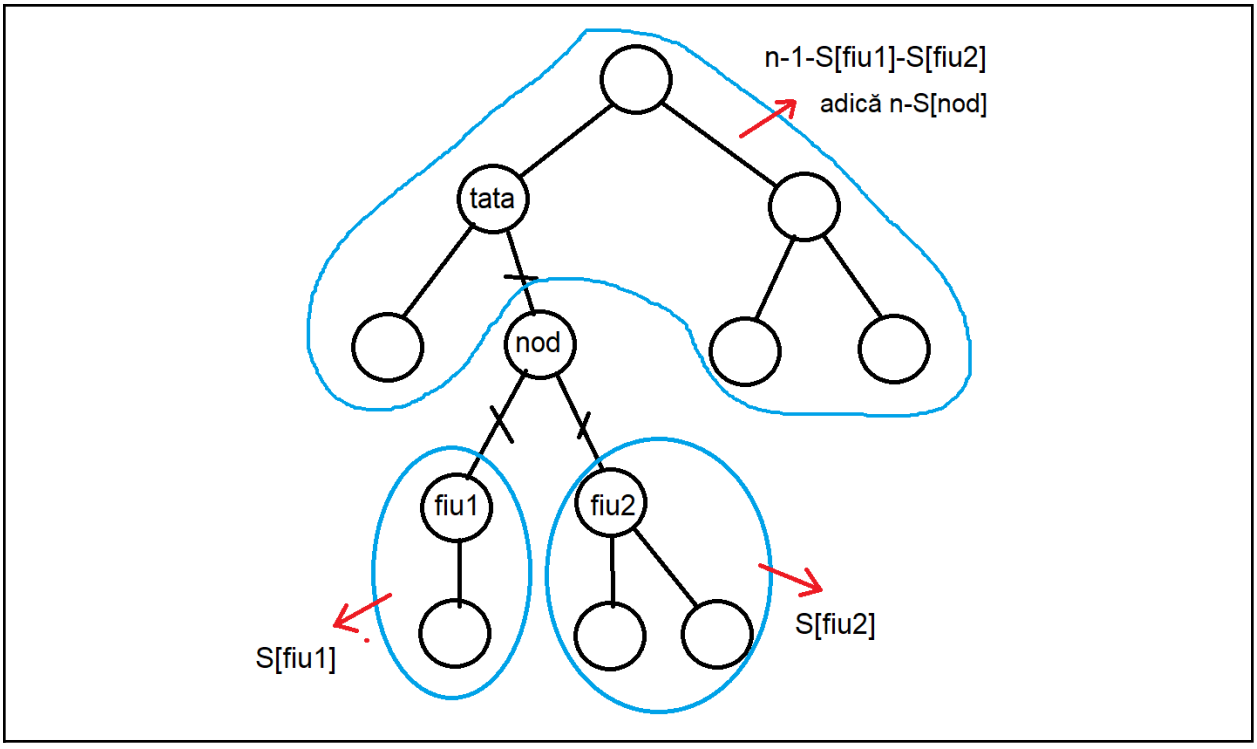
Astfel, la final ne interesează valorile `S` din fii rădăcinii (maximul lor).

Iată mai jos o sursă completă la problema de pe infoarena (nu se obține punctajul maxim întrucât timpul de calcul este de ordin n^2):

```
#include <fstream>
#include <vector>
#define DIM 16010
using namespace std;
vector<int> L[DIM];
int v[DIM];
int S[DIM];
int sol[DIM];
int i, j, minim ,maxim, n, x, y, k;

void dfs(int nod) {
    v[nod] = 1;
    S[nod] = 1;
    for (int i=0;i<L[nod].size(); i++)
        if (v[ L[nod][i]  ] == 0) {
            dfs(L[nod][i]);
            S[nod] += S[ L[nod][i] ];
        }
}
int main() {
    ifstream fin("sediu.in");
    ofstream fout("sediu.out");
    fin>>n;
    for (i=1;i<n;i++) {
        fin>>x>>y;
        L[x].push_back(y);
        L[y].push_back(x);
    }
    minim = DIM;
    for (i=1;i<=n;i++) {
        /// vectorul de vizitați se resetează la
        /// valori nule înainte de fiecare parcurgere
        /// pentru vectorul S nu este necesară resetarea
        /// întrucât lui i se reinițializează valorile înăuntru
        for (j=1;j<=n; j++)
            v[j] = 0;

        dfs(i); /// considerăm fiecare nod drept
                /// rădăcină (cel pe care îl eliminăm)

        /// Determinăm maximul valorilor S pentru fii lui i
        /// Acești fii sunt componentele listei de vecini ai
        /// lui i
        maxim = 0;
        for (j=0;j<L[i].size(); j++)
            if (S[ L[i][j] ] > maxim)
                maxim = S[  L[i][j]  ];
```

```
        if (maxim < minim) {
            minim = maxim;
            k = 1;
            sol[k] = i;
        } else
            if (minim == maxim)
                sol[++k] = i;
    }
    fout<<minim<<" "<<k<<"\n";
    for (i=1;i<=k;i++)
        fout<<sol[i]<<" ";
    return 0;
}
```

Soluția optimă.
Să analizăm mai bine ce se întâmplă dacă facem o singură parcurgere dfs, dintr-un nod oarecare. Mai sus am arătat că putem determina costul obținut prin eliminarea nodului de pornire. Din fericire, cu câteva observații, putem determina în cadrul unei singure parcurgeri costul eliminării pentru oricare nod.
Considerăm un nod care nu este rădăcina. Și pentru el dispunem de valorile `S` calculate în fii săi. Fiecare astfel de valoare este informația pentru o componentă conexă care se rupe *în jos* din el. Mai rămân însă noduri legate prin intermediul tatălui său. Observăm însă că toate aceste noduri formează o singură componentră conexă. Mai mult, numărul de noduri ale sale este ușor de aflat: numărul total de noduri `- 1` (nodul curent) - (suma valorilor `S` din fii nodului curent); sau mai simplu, `n - S[nod]` (pentru că în `S[nod]` se cuprinde și suma din fiii săi și el însuși).
Să analizăm următorul desen:



Sursa de `100p` de pe infoarena.ro

```
#include <fstream>
#include <vector>
#define DIM 16010
using namespace std;

vector<int> L[DIM];
int v[DIM];
int S[DIM];
int sol[DIM];
int Max[DIM];

int i, j, minim, n, x, y, k;

int maxim(int a, int b){
    if (a > b)
        return a;
    else
        return b;
}

void dfs(int nod) {
    v[nod] = 1;
    S[nod] = 1;
    Max[nod] = 0;
    for (int i=0;i<L[nod].size(); i++)
        if (v[ L[nod][i]  ] == 0) {
            dfs(L[nod][i]);
            // cand revin din fiecare fiu
            S[nod] += S[ L[nod][i] ];
            Max[nod] = maxim (Max[nod],  S[ L[nod][i]  ]  );

        }
    /// numarul de noduri din cc de deasupra
    Max[nod] = maxim (Max[nod], n-S[nod]);
}

int main() {
    ifstream fin("sediu.in");
    ofstream fout("sediu.out");
    fin>>n;
    for (i=1;i<n;i++) {
        fin>>x>>y;
        L[x].push_back(y);
        L[y].push_back(x);
    }

    minim = DIM;
    dfs(1);
```

```
    for (i=1;i<=n;i++)
        if (Max[i] < minim) {
            minim = Max[i];
            k = 1;
            sol[k] = i;
        } else
            if (minim == Max[i]) {
                sol[++k] = i;
            }
    fout<<minim<<" "<<k<<"\n";
    for (i=1;i<=k;i++)
        fout<<sol[i]<<" ";
    return 0;
}
```

Folosim un vector suplimentar `Max` în care `Max[nod]` = costul înlăturării lui `nod`. Imediat după revenirea din autoapelurile în fii luăm în calcul la `Max` valoarea `S[fiu]`, apoi, la finalul analizării tuturor fiilor (deci după forul cu autoapelurile) luăm în calcul și numărul de noduri ale subarborelui de deasupra.

2. **Asmax** (https://www.infoarena.ro/problema/asmax)

Enunțul pe scurt
Având un arbore care are nodurile etichetate cu numere întregi (deci și naturale), se cere să determinăm un subarbore al său care să aibă suma costurilor nodurilor maximă (deci subarborele să fie un subgraf conex al său).

Rezolvare
Ne amintim de problema subsecvenței de sumă maximă într-un șir cu valori întregi și apoi arătăm cum o generalizăm pe un arbore.
La subsecvența de sumă maximă pe șir, la poziția curentă `i` căutam să determinăm suma maximă a unei secvențe care are finalul pe poziția `i`. Deci secvența se termina cu elementul `v[i]`. Și aveam două variante: formăm o secvență doar din elementul `v[i]` sau alipim elementul `v[i]` la cea mai bună secvență terminată pe poziția anterioară. Astfel obținem: `s[i] = max(v[i], s[i-1]+v[i])`. Observăm că se selectează ramura a doua doar dacă `s[i-1] > 0` (am notat cu `s[i]` valoarea sumei maxime a unei secvențe care are finalul pe poziția `i`).
Iată cum facem în cazul unui arbore.
Realizăm o parcurgere dfs și la nodul curent în parcurgere (`nod`) determinăm suma maximă a unui subarbore cu rădăcina în `nod`. Evident că pentru asta trebuie să alegem obligatoriu pe `nod` (așa este definiția dinamicii noastre). Notăm cu `s[nod]` această valoare. Trebuie să alegem deci pe `nod` alături de subarbori cât mai buni pentru fii ai săi. Dacă procesarea pentru `nod` o facem la revenirea din autoapel, noi avem deja calculate valorile `s` pentru fii. Orice submulțime a arborilor cu rădăcina în fii am alege aceștia se vor conecta prin intermediul lui `nod`. Astfel este evident că vom alege subarborii corespunzători fiilor care au valoarea `s` pozitivă. În acest fel vom avea:
$s[nod] = v[nod] + s[fiu_{p1}] + s[fiu_{p2}] + \ldots + s[fiu_{pk}]$, unde valorile $s[fiu_{pi}]$ sunt doar cele pozitive dintre valorile `s` ale fiilor lui `i`.

Sursa

```
#include <fstream>
#include <vector>
#define DIM 16002
using namespace std;

vector<int> L[DIM];
int u[DIM], v[DIM], s[DIM];
int sol, n, x, y;

void dfs(int nod) {
    u[nod] = 1;
    s[nod] = v[nod];

    for (int i=0;i<L[nod].size();i++) {
        int fiu = L[nod][i];
        if (u[fiu] == 0) {
            dfs(fiu);
            if (s[fiu] > 0)
                s[nod] += s[fiu];
        }
    }
    sol = max(sol, s[nod]);
}

int main () {
    sol = -1000000000;
    ifstream fin ("asmax.in");
    ofstream fout("asmax.out");

    fin>>n;
    for (int i=1;i<=n;i++) {
        fin >>v[i];
        sol = max(sol, v[i]);
    }

    for (int i=1;i<n;i++) {
        fin>>x>>y;
        L[x].push_back(y);
        L[y].push_back(x);
    }

    dfs(1);
    fout<<sol;
    return 0;
}
```

3. **Cerere** (https://www.infoarena.ro/problema/cerere)

Enunțul pe scurt
Se dă un arbore în care se cunoaște rădăcina. Unele noduri dețin o informație iar pentru altele informația se poate obține doar dacă o poate deține un strămoș precizat al lor. Pentru fiecare nod se cunoaște dacă deține informația sau, în caz contrar, al câtelea dintre strămoșii săi îi poate furniza informația (pentru un nod, tatăl său este primul strămoș, tatăl acestuia este al doilea strămoș, și tot așa până la rădăcină care este ultimul său strămoș - numerotat deci cu valoarea lungimii lanțului de la nod la rădăcină).
Trebuie să determinăm, pentru fiecare nod, prin câți strămoși trebuie să treacă informația pentru a ajunge la el.

Rezolvare
În timpul unei parcurgeri dfs putem păstra într-un parametru și nivelul nodului curent, notat cu `niv` (pe acesta îl gestionăm apelând în fii cu `1` mai mult). Valoarea acestuia este de fapt numărul de noduri prin care se trece de la rădăcină până la nodul curent. Putem să folosim un vector global și la intrarea în apelul recursiv pentru nodul curent să stocăm în acest vector chiar drumul de la rădăcină până la nodul curent: `s[nod] = niv;`
Astfel, dacă pe noi ne interesează pentru `nod` al k_{nod} - lea strămoș, acesta este chiar nodul cu valoarea `s[ niv- k[nod] ]`. Am notat `k[nod]` = al câtelea stramoș al lui `nod` îi poate livra informația lui `nod`. Astfel, `sol[nod] = 1 + sol[ s[niv- k[nod] ] ]` (asta pentru nodurile care nu dețin singure informația).
Avem deci tot o dinamică pe arbore însă de data aceasta nu o mai facem la revenirea din autoapeluri ci la coborâre, pe vectorul care reține nodurile din stiva de autoapeluri.

Sursa

```
#include <cstdio>
#include <vector>
#define DIM 100001
using namespace std;

int k[DIM];
int t[DIM];
int n,i,r,x,y;
int s[DIM];
int sol[DIM];
vector<int> L[DIM];

void dfs(int nod, int niv){
    s[niv]=nod;
    if (k[nod]==0) {
        sol[nod]=0;
    } else {
        sol[nod]=sol[s[niv-k[nod]]]+1;
    }
    for (int i=0;i<L[nod].size();i++) {
        dfs(L[nod][i], niv+1);
    }
}
```

```
int main(){
    FILE *fin = fopen("cerere.in","r");
    fscanf(fin,"%d",&n);
    for (i=1;i<=n;i++)
        fscanf(fin,"%d",&k[i]);
    for (i=1;i<n;i++){
        fscanf(fin,"%d %d",&x,&y);
        L[x].push_back(y);
        t[y] = x;
    }
/// nodul care nu apare în dreapta între perchile de
/// capete de muchii este radacina
    for (i=1;i<=n;i++)
        if (t[i]==0) {
            r=i;
            break;
        }
    dfs(r,1);
    FILE *fout = fopen("cerere.out","w");
    for (i=1;i<=n;i++)
        fprintf(fout,"%d ",sol[i]);

    return 0;
}
```

4. **Srevni** (https://www.infoarena.ro/problema/srevni)

Enunțul pe scurt
Avem un graf orientat și fiecare nod are asociat un cost. Pentru fiecare nod al grafului trebuie să determinăm costul minim dintre toate nodurile în care putem ajunge în el direct sau indirect (evident, avansând pe arce).

Rezolvare
Soluția brută, de a face parcurgere din fiecare nod, are timp foarte mare de calcul: `n*(n+m)`, deoarece facem `n` parcurgeri dfs.
Observăm că dacă am merge invers pe muchii și dacă am face *mai întâi o parcurgere din nodul de cost minim*, servim cu valoarea lui toate nodurile în care ajungem. Reluăm procedeul pentru următorul nod ca și cost, neservit, dar nu vom mai autoapela într-un nod vizitat la o parcurgere anterioară, întrucât acesta și cele în care se ajunge din el au fost atinse din parcurgerea lansata dintr-un nod cu costul mai mic.
Să rezumăm:

- Inversăm muchiile grafului dat.
- Sortăm nodurile după cost.
- Parcurgem nodurile în ordine crescătoare a costului și lansăm dfs dacă nodul curent nu a fost încă vizitat (în cadrul dfs-ului marcăm toate nodurile în care putem ajunge, etichetându-le cu costul celui din care am pornit).

Neintrând decât o singură dată în orice nod, timpul total de calcul va fi de ordin `n+m`.
Această problemă s-ar fi putut rezolva la fel de bine și cu bfs.
Atragem atenția asupra unui detaliu de implementare a cărui cunoaștere ajută foarte mult. Avem

aici de sortat un vector de structuri. Dacă nu le-am fi declarat referință în antetul fucnției de comnparare, la fiecare apel al său în ea s-ar fi făcut copii ale structurilor date ca parametri. Prin felul în care am procedat noi am făcut să se lucreze direct cu structurile din vector și astfel se economisește timp cu duplicarea (cu cât structura respectivă este mai “grea” cu atât optimizarea noastră are efecte mai bune).

Sursa

```
#include <fstream>
#include <vector>
#include <algorithm>
#define DIM 100001
using namespace std;

struct oras {
    int cod;
    int cost;
};

vector<int> L[DIM];
oras x[DIM];
int n,m,a,b,i,j,val;
int sol[DIM];

int cmp (const oras &a, const oras &b) {
    return a.cost < b.cost;
}

void df(int nod){
    sol[nod]=val;
    for (int i=0; i<L[nod].size(); i++)
        if (sol[ L[nod][i] ]==0)
            df(L[nod][i]);
}

int main(){
    ifstream fin("srevni.in");
    fin>>n>>m;
    for (i=1;i<=n;i++) {
        fin>>x[i].cost;
        x[i].cod=i;
    }
    for (i=1;i<=m;i++) {
        fin>>a>>b;
        L[b].push_back(a);
    }

    sort(x+1,x+n+1,cmp);
    for (i=1;i<=n;i++)
        if (sol[x[i].cod]==0) {
            val=x[i].cost;
```

```
            df(x[i].cod);
        }
    ofstream fout("srevni.out");
    for (i=1;i<=n;i++)
        fout<<sol[i]<<" ";
    return 0;
}
```

5. **Iepuri2** (https://www.infoarena.ro/problema/iepuri2)

Enunțul pe scurt
Se dă un arbore cu rădăcină. Trebuie să asociem nodurilor numere naturale cuprinse între `1` și `k` așa încât fiecare nod să aibă asociată o valoare mai mică decât oricare dintre cele asociate nodurilor din subarborele său. Se cere numărul de arbori astfel etichetați care se pot obține. Doi arbori diferă dacă există două noduri poziționate la fel în ei și etichetate cu valori diferite.

Rezolvare
Aproape că ne-am obișnuit să spunem repede: "*dinamică: `D[nod]` = numărul de arbori cu rădăcina în `nod`*", apoi în `D[raracina]` sa avem solutia. În acest caz ne încurcă puțin faptul că valoarea pe care o asociem unui nod depinde de cele asociate deja nodurilor din subarborele său. Ne dăm seama că ne mai trebuie o informație: ce valori am asociat nodurilor fiu (presupunând că dorim să facem calculele într-o dinamică la revenire). Este cazul tipic de a observa nevoia de a adăuga încă o dimensiune la dinamică și avem:
`D[nod][nr]` = numărul de subarbori cu rădăcina în `nod` și pentru care `nod` este etichetat cu valoarea `nr`.
Astfel soluția va fi `D[radacina][1] + D[radacina][2] + … + D[radacina][k]`.
Să vedem ce se întâmplă când revenim în `nod` după apelul într-un anume fiu al său, notat `fiu`. Pentru a calcula `D[nod][nr]` ne interesează `D[fiu][nr+1], D[fiu][nr+2], … D[fiu][k]`. De fapt ne interesează din partea lui `fiu` suma acestor valori. Dacă avem mai mulți fii, evident că avem un fel de produs cartezian al mulțimilor de soluții provenite de la fiecare fiu în parte. Așadar avem de adunat valorile `D` de la fiecare fiu, separat, și de înmulțit aceste sume:

```
D[nod][nr] = (D[fiu1][nr+1]+D[fiu1][nr+2]+…+D[fiu1][k])*
             (D[fiu2][nr+1]+D[fiu2][nr+2]+…+D[fiu2][k])*
           …*(D[fiuf][nr+1]+D[fiuf][nr+2]+…+D[fiuf][k];
```

În sursa de mai jos noi am făcut și câteva optimizări la cod câștigând o dimensiune la complexitatea în timp. După autoapel calculăm valorile `D[nod][nr]` descrescător după `nr` și astfel ne folosim de suma calculată deja în fiu pentru valori mai mari decât `nr` (facem o sumă parțială - urmăriți variabila `sum` din sursă). Scăpăm astfel de un `for`.

O altă observație care aduce o creștere a vitezei: dacă avem de realizat calcule *modulo*, în cazul sumei, dacă ambii operanzi sunt deja modulo, putem evita folosirea operatorului `%` (care este oarecum lent) printr-o testare (`if`) și eventual scădere a lui `MOD` din rezultat. În cazul produsului acest lucru nu este valabil (ar trebui scăzut `MOD` de mai multe ori).

Sursa

```
#include <fstream>
#include <vector>
#define MOD 30011
using namespace std;

vector<int> L[102];
int v[102], rad[102];
int D[102][102];
int n, k, i, x, y;
void dfs(int nod) {
    v[nod] = 1;
    int nrFii = 0;

    for (int i=1;i<=k; i++)
        D[nod][i] = 1;

    for (int i=0; i<L[nod].size(); i++) {
        int fiu = L[nod][i];
        if (v[fiu] == 0) {
            dfs(fiu);
            nrFii++;
            int sum = 0;
            for (int j=k+1; j>=2; j--) {
                sum += D[fiu][j];
                if (sum >= MOD)
                    sum  -= MOD;
                D[nod][j-1] *= sum;
                D[nod][j-1] %= MOD;
            }
        }
    }
    if (nrFii == 0) {
        for (int i=1;i<=k;i++)
            D[nod][i] = 1;
    }
}

int main () {

    ifstream fin ("iepuri2.in");
    ofstream fout("iepuri2.out");

    fin>>n>>k;

    for (i=1;i<n;i++) {
        fin>>x>>y;
        L[x].push_back(y);
        L[y].push_back(x);
        rad[y] = 1;
    }
```

```
    for (int i=1;i<=n;i++)
        if (rad[i] == 0)
            break;
    dfs(i);
    int sol = 0;
    for (int j=1;j<=k;j++) {
        sol += D[i][j];
        if (sol >= MOD)
            sol -= MOD;
    }
    fout<<sol;
    return 0;
}
```

Craiova, 23 ianuarie 2022