Disjoint Set Union

https://cp-algorithms.com/data_structures/disjoint_set_union.html
(набор от разединени множества)
превод от en: github.com/andy489

Съдържание

Тази статия разглежда структурата от данни Disjoint Set Union или DSU. Често се нарича още Union Find поради двете си основни операции.

Тази структура от данни предоставя следните възможности. Дадени са няколко елемента, всеки от които образува отделно самостоятелно множество. DSU ще има операция за *комбиниране* на всеки две множества и ще може да *определи* в кое множество е определен елемент. Класическата версия предоставя още една трета операция - *създаване* на множество от нов елемент.

Следователно основният интерфейс на тази структура от данни се състои само от три операции:

- **make_set(v)** – създава ново множевтво от нов елемент **v**
- **union_sets(a, b)** – слива две множества (множеството , което съдържа елемент **a** с множеството което съдържа елемент **b**)
- **find_set(v)** – връща представителя (още наричан лидер) на множеството което съдържа елемент **v**. Този представител е елемент от същото множество. Избира се лидер/представител от всяко множество от самата структура от данни (и може да се променя с времето, конкретно след извикване на операцията **union_sets**). Този представител може да се използва за проверка дали два елемента са часто от едно и също множество или не. **a** и **b** са в едно и също множество, ако **find_set(a) == find_set(b)**. В противен случай елементите **a** и **b** са в различни множества.

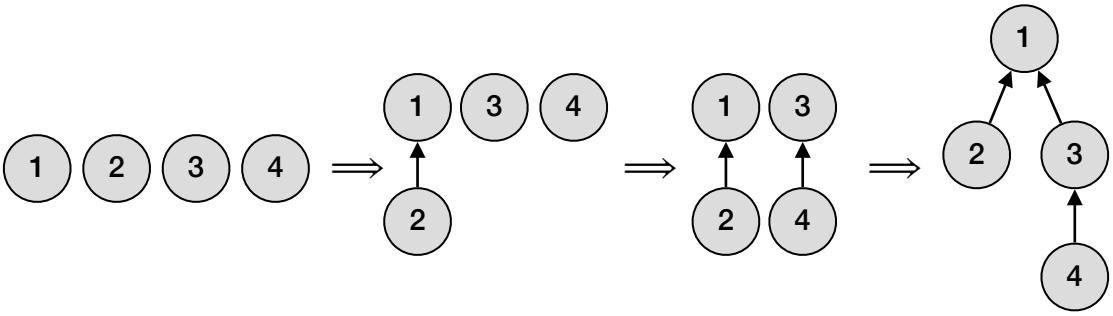
Както ще опишем по-подробно по-късно, структурата от данни ни позволява да извършваме всяка от тези операции за почти $O(1)$ амортизирана времева сложност.

Също така в един от подразделите е обяснена алтернативна структура на DSU, която постига по-бавна амортизирана сложност от $O(\log n)$ за операциите (методите) ѝ, но може да бъде по-мощна от обикновената DSU структура.

Построяване на ефективна структура от данни

Ще съхраняваме множествата под формата на *дървета*: всяко дърво ще отговаря на едно множество. Корена на дървото ще бъде представителя/лидера на множеството.

На следната картинка може да разгледаме представянето на такова дърво.



В началото, всеки елемент образува самостоятелно множество, следователно всеки връх се разглежда като отделно дърво. След което обединяваме множеството съдържащо елемент 1 и множеството съдържащо елемент 2. След това обединяваме множеството съдържащо елемент 3 и множеството съдържащо елемент 4. На последната стъпка обединяваме множеството съдържащо елемент 1 и множеството съдържащо елемент 3.

За да имплементираме тази структура ще трябва да поддържаме масив **parent**, който съхранява референция до неговия непосредствен родител в дървото.

Наивна (неефективна) имплементация

Вече може да напишем първата имплементация на Disjoint Set Union структурата от данни. В началото ще бъде доста неефективна, но по-късно ще я подобрим с помощта на две оптимизации, така че да отнеме почти константно време за всяко извикване на функция/метод на структурата.

Както споменахме, цялата информация за множествата от елементи ще се съхранява в родителски масив **parent**.

За да създадем ново множесто (операция **make_set(v)**), просто създаваме дърво с корен във върха **v**, което означава, че той е негов собствен родител (корена ще е този връх, който е сам родител на себе си).

За обединяването на две множества (операция **union_sets(a, b)**), първо намираме представителя на множеството, което съдържа елемента **a** и представителя на множеството което съдържа елемента **b**. Ако представителите са идентични, тогава няма нужда да се прави нещо - множествата вече са обединени. В противен случай може просто да уточним, че единят от представителите е родител на другия представител - по този начин съчетаваме двете дървета.

И накрая, имплементацията на функцията за намиране на представител на множеството в което се намира даден елемент (операцията **find_set(v)**): просто се изкачваме по родителите на върха **v** докато не стигнем корена, т.е. такъв връх, за който референцията към родителя му сочи към самия него. Тази операция лесно може да се имплементира рекурсивно.

```
void make_set(int v) {
    parent[v] = v;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b)
        parent[b] = a;
}
```

Тази имплементация обаче не е ефективна. Лесно е да се изгради пример, в който дърветата са изродени (свързани списъци) и се разпадат на много дълги вериги. В този случай всяко едно извикване на **find_set(v)** ще отнеме $O(n)$ линейна времева сложност.

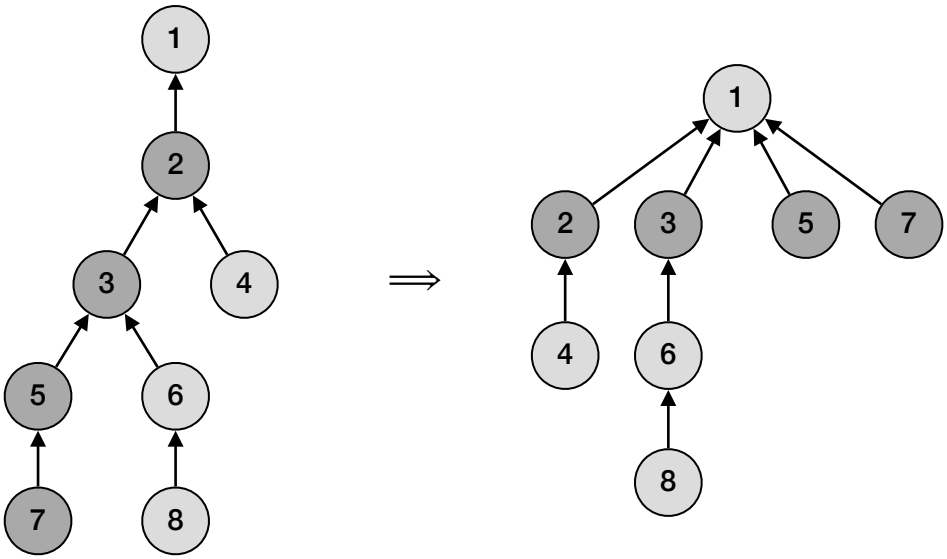
Тази сложност е твърде далеч от сложността, която искаме да имаме (почти константна по време). За това ще разгледаме две оптимизации, които ще позволят значително да се ускори работата.

Оптимизация за компресиране на пътя

Тази оптимизация е предназначена за ускоряване на **find_set**.

Ако извикаме **find_set(v)** за някакъв връх **v**, всъщност намираме представителния връх **p** за всики върхове, които сме посетили по пътя между **v** и действителния представител **p**. Трикът е да се направят пътищата за всички тези върхове по-кратки, като се зададе родителят на всеки посетен връх директно на **p**.

Следното изображение онагледява описаната операция. От лявата страна има дърво, а от дясната страна е компресираното дърво след извикване на **find_set(7)**, което съкращава пътеките за посетените възли 7, 5, 3 и 2.



Новата имплементация на **find_set** е следната:

```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

Простата реализация прави точно това, което планирахме: първо намира представителя на множеството (корена на дървото), а след това в процеса на развиване на рекурсията – посетените върхове се прикачват директно към намерения представител.

Тази проста модификация на операцията вече постига амортизирана времева сложност $O(\log n)$ за извикване (link). Има и втора модификация, която ще я направи дори по-бърза.

Обединение по размер/ранг

В тази оптимизация ще променим операцията **union_set**. За да бъдем точни, ще променим кое дърво се привързва към другото. В естественото изпълнение второто винаги се привързва към първото. на практика това може да доведе до дървета, съдържащи вериги с дължина $O(n)$. С тази оптимизация ще избегнем това, като изберем много внимателно кое дърво да се прикачи.

Има много възможни евристики, които могат да се използват. Най-популярни са следните два подхода: В първия подход използваме размера на дърветата като ранг, а във втория използваме дълбочината на дървото (по-точно горната граница на дълбочината на дървото, защото дълбочината ще стане по-малка когато приложим компресия на пътя)

И в двата подхода същността на оптимизацията е една и съща: ние прикрепяме дървото с по-нисък ранг към това с по-голям ранг.

Имплементацията на обединение по размер:

```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

Имплементация на обединение по ранг, базиран на дълбочината на дърветата:

```
void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets(int a, int b) {
```

```
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

И двете оптимизации са еквивалентни по отношение на времева сложност и сложност по памет.

Времева сложност

Както споменахме по-горе, ако комбинираме и двете оптимизации - компенсиране на пътя с обединяване по размер/ранг - ще достигнем почти константни заявки във времето. Оказва се, че крайната амортизирана времева сложност е $O(\alpha(n))$, където $\alpha(n)$ е обратната функция на Акерман, която расте много бавно. Всъщност тя расте толкова бавно, че не надвишава 4 за всички разумни n (приблизително $n < 10^{600}$).

Амортизирана сложност е общото време на операция, оценено в последователност от множество операции. Идеята е да се гарантира общото време на цялата последователност, като същевременно се позволява отделните операции да са много по-бавни от амортизираното време. Например в нашия случай едно извикване може да отнеме $O(\log n)$ в най-лошия случай, но ако направим m такива извиквания след това, ще завършим със средно време $O(\alpha(n))$.

Доказателството за тази сложност по време е доста дълга и сложна, за това няма да се занимаваме с нея в тази статия.

Също така си струва да споменем, че DSU с обединение по размер/ранг, но без компресия на пътя работи в $O(\log n)$ времева слоожност за заявка.

Свързване чрез индекси/свързване на принципа на хвърляне на монета

Както обединението по ранг, така и обединението по размер изискват да съхраняваме допълнителни данни за всяко множество и да поддържаме тези стойности по време на всяка операция на обединение. Съществува обаче и рандомизиран алгоритъм, който малко опростява операцията на съюз: свързване по индекс.

Присвояваме на всяко множество произволна стойност, наречена индекс, и прикрепяме множеството с по-малкия индекс към този с по-големия. Вероятно е по-голямо множество да има по голям индекс от по-малкиото, следователно тази операция е тясно свързана с обединението по размер. Всъщност може да се докаже, че тази операция има същата сложност във времето като обединението по размер. Но на практика това е малко по-бавно от обединението по размер.

Може да намерите доказателство за сложността и дори повече техники на обединение тук: https://www.cis.upenn.edu/~sanjeev/papers/soda14_disjoint_set_union.pdf

```
void make_set(int v) {
    parent[v] = v;
    index[v] = rand();
}
```

```
void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (index[a] < index[b])
            swap(a, b);
        parent[b] = a;
    }
}
```

Приложения и различни подобрения

В този раздел ще разгледаме няколко приложения на структурата от данни DSU, както тривиалните, така и някои подобрения на структурата.

Свързани компоненти в граф

Това е едно от очевидните приложения на DSU.

Формално проблемът е дефиниран по следния начин: Първоначално имаме празен граф. Трябва да добавим върхове и ненасочени ребра и да отговаряме на запитвания от вида (a, b) - „върховете a и b в една и съща компонента на свързаност от графа се намират?“

Тук може директно да приложим структурата от данни и да получим решение, което обработва добавянето на връх или ребро и заявка в приблизително константно време в средния случай.

Това приложение е доста важно, тъй като почти същия проблем се появява в алгоритъма на Kruskal за намиране на минимално покриващо дърво (https://cp-algorithms.com/graph/mst_kruskal.html). С помощта на DSU може да подобрим (https://cp-algorithms.com/graph/mst_kruskal_with_dsu.html) времевата сложност от $O(m \log n + n^2)$ до $O(m \log n)$.

Търсене на свързани компоненти в снимка

Едно от приложенията на DSU е следната задача: имаме изображения с $n \times m$ пиксела. Първоначално всички са бели но след това се рисуват няколко черни пиксела. Искаме да определим размера на всеки свързан бял компонент в крайното изображение.

За решението ние просто итерираме над всички бели пиксели в изображението, за всяка клетка се повтаря итерацията над четирите ѝ съседи и ако съседът е бял, извикваме **union_sets**. Така ще имаме DSU с nm възли, съответстващи на пикселите на изображението. Получените дървета в DSU са желаните свързани компоненти.

Проблемът може да бъде решен и от DFS (https://cp-algorithms.com/graph/depth-first-search.html) или BFS (https://cp-algorithms.com/graph/breadth-first-search.html) алгоритъм, но описаният тук метод има предимство: той може да обработва матрицата ред по ред (т.е. за обработка на ред имаме нужда само от предишния и текущия ред и се нуждаем само от изградена DSU за елементите на един ред) в $O(\min(n, m))$ памет.

Съхраняване на допълнителна информация за всеки комплект/множество

DSU ни позволява лесно да съхраняваме допълнителна информация за всяко множество. Прост пример е размера на множеството: запазването на размерите беше вече описано в раздела „Обединение по размер/ранг“ (информацията беше запазена в текущия представител на множеството).

По същия начин - като я съхраняваме в представителните върхове - може да съхраняваме всяка друга информация за множествата.

Компресирано сегментно обхождане /оцветяване на подмасиви (offline)

Едно често приложение на DSU е следното: Съществува набор от върхове и всеки връх има изходящо ребро към друг връх. С DSU можем да намерим крайната точка, до която стигаме след като следваме всички ребра от дадена начална точка, в почти константно време.

Добър пример за това приложение е **проблемът с боядисването на подмасиви**. Имаме сегмент с дължина L, всеки елемент първоначално има цвят 0. Трябва да пребоядисаме подреда $[l, r]$ с цвета c за всяка заявка (l, r, c). В края искаме да намерим финалния цвят на всяка клетка. Допускаме, че знаем всички заявки предварително, т.е. задачата е offline.

За решението може да направим DSU, която за всяка клетка да съхранява връзката към следващата неоцветена клетка. Така първоначално всяка клетка ще сочи към себе си. След боядисване на сегмент, всички клетки от този сегмент ще сочат към клетката след сегмента.

Сега, за да решим този проблеми, ние разглеждаме заявките **в обратен ред**: от последната до първата. По този начин, когато изпълняме заявка, трябва само да боядисаме точно неизрисуваните клетки в подредицата $[l, r]$. Всички останали клетки вече съдържат окончателния си цвят. За бърза итерация върху всички неоцветени клетки използваме DSU. Намираме най-лявата неоцветена клетка вътре в сегмент, пребоядисваме я и с показалеца се преместваме в следващата празна клетка вдясно.

Тук може да използваме DSU с компресия на пътя, но не може да използваме компресия по ранг/размер (защото е важно кой става лидер след сливането). Следователно сложността ще бъде $O(\log n)$ за обединение (което също е доста бързо).

Имплементация:

```
for (int i = 0; i <= L; i++) {
    make_set(i);
}

for (int i = m-1; i >= 0; i--) {
    int l = query[i].l;
    int r = query[i].r;
    int c = query[i].c;
    for (int v = find_set(l); v <= r; v = find_set(v)) {
        answer[v] = c;
        parent[v] = v + 1;
    }
}
```

Има една оптимизация: Можем да използваме **обединение по ранг**, ако съхраняваме следващата неоцветена клетка в допълнителен край end[ ] масив. Тогава може да обединим две множества в едно - класирано според тяхната евристика и получаваме решението в $O(\alpha(n))$.

Поддържане на разстояния до представител на комплект

Понякога в конкретни приложения на DSU трябва да поддържате разстоянието между върха и представителя на неговия набор (т.е. дължината на пътя в дървото от текущия възел до корена на дървото).

Ако не използваме компресия на пътя, разстоянието е точно броя на рекурсивните извиквания. Но това ще бъде неефективно.

Възможно е обаче да се направи компресия на пътя, ако съхраним разстоянието до родителя като допълнителна информация за всеки възел.

В имплементацията е удобно да се използва масив от двойки за **parent[ ]** и функцията **find_set** да връща две числа: представителя на множеството и разстоянието до него.

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
}

pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int len = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second += len;
    }
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a).first;
    b = find_set(b).first;
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = make_pair(a, 1);
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

Поддържане на паритета на дължината на пътя / Проветка на двустранността (online)

По същия начин като изчисляването на дължината на пътя към лидера, е възможно да се поддържа паритета на дължината на пътя пред него. Защо това приложение е в отделен параграф?

Необичайното изискване за съхранение на паритета на пътя се появява в следната задача: първоначално ни се дава празен граф. Към него могат да се добавят ребра и трябва да отговаряме на заявки от вида „съдържа ли свързаният компонент този връх двустранно?“.

За да разрешим този проблем, правим DSU за съхранение на компонентите и съхраняваме паритета на пътя до представителя за всеки връх. По този начин може бързо да проверим дали добавянето на ребро води до нарушение на двустранността или не: а именно, ако краищата на реброто лежат в един и същ свързан компонент и имат еднаква дължина на

паритет към лидера, то тогава добавянето на това ребро ще доведе до цикъл с нечетна дължина и компонентът ще загуби свойството на двустранност.

Единствената трудност, пред която сме изправени, е да изчислим паритета в метода **union_find**.

Ако добавим ребро (a, b), което свързва две свързани компоненти в една, то тогава когато прикачваме едното дърво към другото ни е необходимо да пренастройм паритета.

Нека изведем формулата, която изчислява паритета, издаден на лидера на множеството, който ще се прикачи към друго множество. Нека x бъде четността на дължината на пътя от върха a до своя лидер A и y е четността на дължината на пътя от върха b до своя лидер B, а t желания паритет (четност), която искаме да съхраним за B след сливането/обединението. Пътеката съдържа три части: от B до b, от b до a, които са свързани с едно ребро и следователно има паритет 1, и от a до A. Следователно получаваме формулата ($\oplus$ обозначава операцията XOR):
$t = x \oplus y \oplus 1$.
По този начин, независимо от това колко съединяния извършваме, четността на ребрата се пренася от лидер към друг.

Ще покажем реализация на DSU, която поддържа паритет. Както в предишния раздел, ние използваме двойка, за да съхраняваме прародителя и паритета. В допълнение за всеки набор съхраняваме в масива **biparitite[ ]** независимо дали той все още е двустранен или не.

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}

pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int parity = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second ^= parity;
    }
    return parent[v];
}

void add_edge(int a, int b) {
    pair<int, int> pa = find_set(a);
    a = pa.first;
    int x = pa.second;

    pair<int, int> pb = find_set(b);
    b = pb.first;
    int y = pb.second;

    if (a == b) {
        if (x == y)
            bipartite[a] = false;
    } else {
        if (rank[a] < rank[b])
            swap (a, b);
```

```
        parent[b] = make_pair(a, x^y^1);
        bipartite[a] &= bipartite[b];
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

bool is_bipartite(int v) {
    return bipartite[find_set(v).first];
}
```

RMQ (range minimum query) минимална заявка за обхват за $O(\alpha(n))$ амортизирана сложност (offline) / Трик на Arpa

Даден ни е масив **a[ ]** трябва да изчислим някои минимуми в дадени сегменти от масива.

Идеята за решаване на този проблем с DSU е следната: Ще направим повторение над масива и когато сме на **i**-тия елемент, ще отговорим на всички запитвания **(L, R)** с **R == i**. За да направим това ефективно, ще запазим DSU, използвайки първите **i** елемента със следната структура: родителят на елемент е следващия по-малък елемент вдясно от него. След това, използвайки тази структура, отговорът на заявка ще бъде **[find_set(L)]**, най-малкото число вдясно от **L**.

Този подход очевидно работи само offline, т.е. ако предварително знаем всички заявки.

Лесно е да се види, че може да се приложи компресия на пътя. И може също да използваме обединение по ранг, ако съхраняваме действителния лидер в отделен масив.

```
struct Query {
    int L, R, idx;
};

vector<int> answer;
vector<vector<Query>> container;
```

container[i] съдържа всички заявки, за които **R == i**

```
stack<int> s;
for (int i = 0; i < n; i++) {
    while (!s.empty() && a[s.top()] > a[i]) {
        parent[s.top()] = i;
        s.pop();
    }
    s.push(i);
    for (Query q : container[i]) {
        answer[q.idx] = a[find_set(q.L)];
    }
}
```

В днешно време този алгоритъм е известен като трик на Arpa. Той е кръстен на AmirReza Poorakhavan, който независимо откри и популяризира тази техника.

LCA (lowest common ancestor in a tree) най-близък общ прародител за $O(\alpha(n))$ амортизирана сложност

Алгоритъмът за намиране на LCA е разгледан в статията **Lowest Common Ancestor – Tarjan's off-line algorithm**: https://cp-algorithms.com/graph/lca_tarjan.html. Този алгоритъм се конкурира благоприятно с други алгоритми за намиране на LCA поради неговара простота (особено в сравнение с оптимален алгоритъм като този от **Farach-Colton and Bender**: https://cp-algorithms.com/graph/lca_farachcoltonbender.html)

Съхраняване на DSU изрично в списък със задачи / Приложения на тази идея при обединяване на различни структури от данни

Един от алтернативните начини за съхранение на DSU е запазването на всяко множество под формата на **изрично съхранен списък на неговите елементи**. В същото време всеки елемент съхранява и препратка към прародителя на своето множество.

На пръв поглед това изглежда като неефективна структура от данни: като комбинираме две множества, ще трябва да добавим един списък в края на друг и трябва да актуализираме лидерството във всички елементи на един от списъците.

Оказва се обаче, че използването теглова евристика (подобна на обединяването по размер) може значително да намали асимптотичната сложност: $O(m + n \log n)$ за да изпълни m заявки за n елемента.

Под теглова евристика разбираме, че винаги ще добавяме по-малкото от двете множества към по-голямото. Добавянето на един набор към друг е лесно да се приложи в **union_sets** и ще отнеме време, пропорционално на размера на добавеното множество. А търсенето на лидера във **find_set** ще отнеме $O(1)$ с този метод на съхранение.

Нека докажем сложността по време $O(m + n \log n)$ за изпълнение на m заявки. Ще фиксираме произволен елемент x и ще броим колко често е докосван в операцията по сливане **union_sets**. Когато елементът x се докосне от първия път, размерът на новото множество ще бъде поне 2. Когато се докосне втори път, полученото множество ще има размер поне 4, защото по-малкото множество се добавя към по-голямото. И така нататък. Това означава, че x може да се премести само най-много $\log n$ пъти от операции по сливане. Така сумата над всички върхове дава $O(n \log n)$ плюс $O(1)$ за всяка от m-те заявки.

Ето и имплементация:

```
vector<int> lst[MAXN];
int parent[MAXN];

void make_set(int v) {
    lst[v] = vector<int>(1, v);
    parent[v] = v;
}

int find_set(int v) {
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a);
```

```
    b = find_set(b);
    if (a != b) {
        if (lst[a].size() < lst[b].size())
            swap(a, b);
        while (!lst[b].empty()) {
            int v = lst[b].back();
            lst[b].pop_back();
            parent[v] = a;
            lst[a].push_back (v);
        }
    }
}
```

Идеята за добавяне на по-малката част към по-голямата може да се използва и в много решения, които нямат нищо общо с DSU.

Например за следния проблем: получаваме дърво, на всяко листо има определен номер (един и същ номер може да се появи многократно на различни листа). Искаме да изчислим броя на различни числа в поддървото за всеки възел на дървото.

Прилагайки към тази задача същата идея, е възможно да се получи това решение: може да реализираме DFS, което ще върне показалец към множество от цели числа - списъка с числа в това поддърво. След това, за да получим отговора за текущия връх (освен ако разбира се не е листо), извикваме DFS за всички деца на този връх и обединяваме всички получени множества заедно. Размерът на полученото множество ще бъде отговорът за текущия връхв. За ефективно комбиниране на множества просто прилагаме гореописаната рецепта: обединяваме като добавяме по-малки към по-големи множества. В крайна сметка получаваме $O(n \log n)$ решение, тъй като едно число ще бъде добавено само към множество от най-много $O(\log n)$ пъти.

Съхраняване на DSU чрез поддържане на ясна структура на дърво / намиране на мост в $O(\alpha(n))$ амортизирана сложност (online)

Едно от най-мощните приложения на структурата от данни DSU е, че позволява да съхраняваме както компресирани, така и некомпресирани дървета. Компресираната форма може да се използва за сливане на дървета и за проверка, ако върховете са в едно и също дърво, а некомпресираната форма може да се използва - например - за търсене на пътища между два дадени върха или други обиколко на структурата на дървото.

В реализацията това означава, че в допълнение към компресирания родителски масив от **parent[ ]** ще трябва да запазим масива от некомпресирани прародители **real_parens[ ]**. Тривиално е, че този допълнителен масив няма да влоши сложността: промените в него настъпват само когато обединим две дървета и то само в един елемент.

От друга страна, когато се прилага на практика, често се налага да свързваме дървета, като използваме определено ребро, вместо да използваме двата корена на дърветата. Това означава, зе нямаме друг избор, освен да възстановим отново едно от дърветата (ще направим краищата на реброто новите корени на дърветата).

На пръв поглед изглежда, че това повторно вкореняване е много скъпо и значително ще влоши времевата сложност. Наистина, за вкореняване на дърво във връх v трябва да преминем от върха към стария корен и да променим посоките в **parent[ ]** и **real_parent[ ]** за всички възли по този път.

Но в действителност не е толкова лошо, може просто да възстановим по-малкото от двете дървета, с подобни на идеите на предишните раздели ще получим $O(\log n)$ амортизирана сложност.

Историческа ретроспекция

Структурата от данни DSU е известна от доста време.

Този начин на съхранение на тази структура под формата на **гора от дървета** е описан за първи път от Galler и Fisher през 1964 г. (Galler, Fisher, "An Improved Equivalence Algorithm). Но пълният анализ на сложността на времето е извършен много по-късно.

Компресията на пътя за оптимизация и обединението по ранг е разработена от McIlroy и Morris и независимо от тях също от Tritter.

Hopcroft и Ullman показват през 1973 г. сложността по време от $O(\log^\star n)$ (Hopcroft, Ullman "Set-merging algorithms") – тук $O(\log^\star n)$ е **iterated logarithm** (това е бавнорастяща функция, но все още не толкова бавна, колкото обратната функция на Акерман).

За първи път оценката на $O(\alpha(n))$ е показана през 1975 г. (Tarjan "Efficiency of a Good But Not Linear Set Union Algorithm"). По-късно през 1985 г., той заедно с Leeuwen публикува множество анализи на сложността за няколко различни ранг евристики и начини за компресиране на пътя (Tarjan, Leeuwen "Worst-case Analysis of Set Union Algorithms").

Накрая през 1989 г. Fredman и Sachs доказват, че във възприетия модел на изчисление **всеки** алгоритъм за Disjoint Set Union проблема, трябва да работи в за поне $O(\alpha(n))$ време по сложност в средния случай (Fredman, Saks, "The cell probe complexity of dynamic data structures").

Освен това трябва да се отбележи, че има няколко статии, оспорващи тази времева оценка и твърдящи, че DSU с компресия на пътя и обединение по ранг работи в $O(1)$ константно време в средния случай (Zhang "The Union-Find Problem Is Linear", Wu, Otoo "A Simpler Proof of the Average Case Complexity of Union-Find with Path Compression").