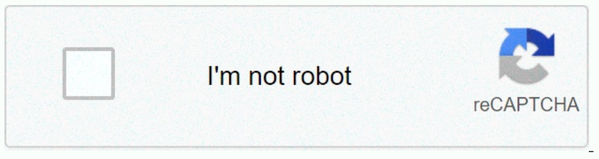

Continue

Disjoint set data structure

MAKE-SET(x) creates a new set, the only member of which is assigned x; Note that x is not in other series. UNION(x,y) combines two dynamic sets containing objects x and y, such as Sx and Sy, into a new series with Sx ∪ Sy , stating that Sx ∩ Sy =∅; FIND-SET(x) returns the pointer x. INSERT(a,S) parameter to a delegate, adds object A to S, and returns S∪{a}. DELETE(a,S) deletes object A from S and returns the S-{a}. SPLIT(a,S) divides S objects into two series S1 and S2 so that S1 ={b | b≤a & b∈S}, and S2 =S-S1 . MINIMUM(S) returns the minimum object of the S. Connected Components (CC) Minimum Spanning Trees (MSTs) CONNECTED-COMPONENTS (G) 1 function for each v∈V 2 do MAKE-SET (v) 3 for each edge (u,v)∈E).4 to make, if FIND-SET (u) ≠ FIND-SET (v) 5 then UNION (u,v) SAME-COMPONENTS (u,v) 1 if FIND-SET (u) = FIND-SET (v) 2, return TRUE 3 returns false join according to order. path compression MAKE-SET (x) 1 p[x] ← x 2 rank[x] ← 0 FIND-SET (x) 1 if x≠p[x] 2 then p[x] ← FIND-SET (p[x]) 3 return p[x] UNION (x,y) 1 LINK(FIND-SET (x), FIND-SET (y)) LINK (x,y) 1 if rank[x]>rank[y] 2 then p[y] ← x 3 else p[x] ← y 4 if rank[x]=rank[y] 5 then rank[y] ← rank[y]+1 rank[x]≤rank[p[x]] for any tree root x, size(x)≥ 2rank[x] (Link operation) for any integer r, there are at most n/ 2r nodes of rank r each node has rank at most ⌊logn⌋, assuming there are at n objects involved. This article discusses the data structure Disjoint Set Union or DSU. It is also often called Union Find because of its two main operations. This data structure includes the following properties. We are given several elements, each of which is a separate series. DSU has a function that combines two sets and can tell which particular element a particular element is in. The classic version also introduces the third function, it can create a set from a new element. Thus, the basic interface of this data structure consists of only three functions: make_set(v) - creates a new set consisting of the new element v union_sets(a, b) - combines two specified set (the set in which element a is located, and the set in which element b is located) find_set(v) - returns the representative of the set that contains element v (also called the leader). This representative is part of a similar group. The data structure selects it in each set (and may change over time, i.e. union_sets This delegate can be used to check find_set two elements are part of the same find_set or not. Otherwise, they're in different leagues. As described later, the data structure allows you to make each of these activities almost $O(1)$ on average. Also, one sub-section explains the alternative structure of DSU, which achieves a slower average complexity of $O (\log n)$, but can be more effective than a standard DSU structure. Build a powerful data structure We store the series in the form of trees: each tree corresponds to a single set. And the root of the tree becomes the representative/manager of the set. In the following image, you will see a representation of such trees. In the beginning, each element begins as a single set, so each vertage is its own tree. Then we combine a set with element 1 and a set with element 2. Then we combine a set with element 3 and a set with element 4. And in the final step, we combine a set with element 1 and a set containing element 3. For implementation, this means that we must maintain the main level of the table, which stores a reference to its immediate ancestor tree. Naïve implementation We can already write the first implementation of the Union's information structure. It's pretty inefficient at first, but later we can improve it with two optimizations to take up almost standard time for each function call. As we said, all information about element sets is stored in the main set of the table. To create a new set (function make_set(v)), we create a tree with a root folder at point v, which means it is its own ancestor. If you want to combine two sets (union_sets(a, b)), we first find the representative of the set where a is located and the representative of the part where b is located. If the delegates are identical that we have nothing to do, the series are already combined. Otherwise, we can simply specify that one of the representatives is the parent of the other representative - and thus combines the two trees. Finally, to find a representative function (Operation find_set(v)): we climb only the ancestors of vertile v until we reach the root, that is, the vertage that leads to itself. This function is easy to perform recursively. 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) older[b] = a; } However, implementation is ineffective. It is easy to build an example so that trees degenerate into long chains. In this case, each find_set(v) can $O(n)$ time. It's a long way off. the complexity we want (almost standard time). That is why we are considering two optimizations to significantly speed up work. Path compression optimization This optimization is designed to speed up find_set. If we call find_set(v) some vert point v, we will actually find a representative p for all the vert points we visit in the path of v and the actual representative p. The trick is to shorten the paths of all those knots by placing the main level at each of the visited vert points directly at p. The operation is shown in the following illustration. On the left is a tree, and on the right side there is a pressed tree after find_set(7), which shortens the paths of the visited nodes 7, 5, 3 and 2. Find_set's new implementation is as follows: int find_set(int v) { if (v == parent[v]) return v; return parent[v] = find_set(parent[v]); } Simple execution does what was meant to do: first find the representative (root point) of the set, and then, when the stack erupts, the nodes that visited are connected directly to the delegate. This simple change in functionality already reaches the time consency $O(\log n)$ per call on average (here without evidence). There is another change that makes it even faster. According to the size/ranking of the Union In this optimization, we union_set our operations. More specifically, we're going to change which tree attaches to the other. In naïve implementation, the second tree was always attached to the first. In practice, it can lead to trees that contain $O(n)$. With this optimization, we avoid this by choosing very carefully which tree attaches. There are many possible heurists that can be used. The most popular are the following two approaches: in the first approach, we use the size of the trees as an investment, and in the second we use the depth of the tree (more precisely, the upper limit of the depth of the tree, since the depth decreases when using the compression of the path). In both approaches, the essence of optimization is the same: we attach the lower value of the tree to the one with a higher value. Here is the Union's implementation by size: make_set(int v) { parent[v] = v; size[v] = 1; } blank 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); older[b] = a; size[a] += size[b]; } } And here is the implementation of the covenant according to order based on the depth of the trees: make_set(int v) { older[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]++; } Both optimizations correspond to the complexity of time and space. So practically you can use any of them. The complexity of time As mentioned earlier if we combine both - packing the path according to size/ranking - we reach almost continuous time polls. It turns out that the final accrued time complexity is $O(\alpha(n))$, where $\alpha(n)$ is an inverse Ackermann function that grows very slowly. In fact, it grows so slowly that it doesn't exceed $4 for all reasonable n (about $$n < $10 {600}$). Accrued complexity is the total operation-specific time that is evaluated during a series of multiple operations. The idea is to guarantee the total time of the entire series while allowing one operation to be much slower than the amortized time. For example, in our case, one call can, in the worst case scenario, take $O (\log n)$, but if we make $$m of such calls back, we will end up with an average time of $O(\alpha(n)$. Nor do we provide evidence of the complexity of this time, as it is quite long and complex. It is also worth mentioning that DSU, which has a union by size/ranking, but without path compression works $O (\log n)$ time per survey. To link using an index/coin flip that combines both union ranking and join size, you must save additional information about each series and keep these values during each join operation. There is also a randomized algorithm that simplifies the union's operations a little: linking with an index. We assign a random value to each set, called an index, and assign a set selected with a smaller index to the one with a larger index. It is likely that a larger series has a larger index than a smaller series, so this function is closely related to the union by size. In fact, it can be shown that this operation has the same time complexity as the union by size. In practice, however, it is slightly slower than the union. Here you will find proof of complexity and even more of union techniques. 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; } } It is a common misconception that flipping a coin, deciding which set we attach to another, has the same complexity. But that is not true. The paper linked above, according to guesswork, coin flip link combined with path compression is complex $\Omega\left(n \frac{\log n}{\log \n}\right)$. And in benchmarks, it works much worse than a union through size/ranking or index linking. void union_sets(int a, int b) { a = find_set(a); b = find_set(b) if (a != b) { if (rand() %2) swap(a, b); parent[b] = a; } Applications and various improvements This section looks at multiple applications in the data structure, both trivial uses, and some improvements to the data structure. Components in the diagram This is one of the obvious applications of DSU. Formally, the problem is defined as follows: At first, we have an empty chart. We need to increase the top bonds and the directional edges. Edges. inquiries about the form (a, b) - are the top a and $$b in the same combined component chart? Here, we can directly apply the data structure and get a solution that handles the vert tip or edge and the addition of the survey within an average of almost standard time. This application is quite important because the krushal algorithm shows almost the same problem to find the minimum size tree. With DSU, we can improve $O(m \log n + n^2)$ $O(m \log n)$. Finding mapped components in an image One of DSU's applications is the following task: the image is $n \times m$ pixels. Originally, everyone is white, but then a few black pixels are drawn. You want to specify the size of each white merged portion of the final picture. To find a solution, simply iterate all the white pixels in the image, iterate iterated for each cell over its four neighbors, and if the neighbor is white, call union_sets. This way, we have a DSU with $n m$ knots that match the image pixels. The resulting trees in the DSU are the desired connected components. The problem can also be solved by DFS or BFS, but the method described here has an advantage: it can process the matrix line by line (i.e. deal with a row that we only need the previous and current row, and you only need one DSU built for the elements in the elements) $O(\min(n, m))$ in memory. Save more information about each series of DSU, which allows you to easily store additional information in sets. A simple example is the size of the sizes: the recording of sizes in the Union was already described according to the size section (the data was recorded by the current representative of the set). In the same way - by storing its representative nodes - you can also store other information about the set. Pack jumps along segment/Painting subarrays offline One common application of DSU is as follows: The vertex set is a set, and each vertex has an outbound edge at the other vertex. With DSU, you can find an end point that we can get to after following all the edges from the given starting point, almost in standard time. A good example of this application is the problem of painting subprocesses. We have a length segment L, each element initially has a color of 0. We need to paint a subrray with $[l, r]$ color c for each query (l, r, c). At the end, we want to find the ultimate color of each cell. We assume that we know all queries in advance, that is, the task is offline. For the solution, we can make a DSU, which for each cell stores a link to the next unblemished cell. Thus, at first, each cell points to itself. After you re-paint one of the requested segments, all cells in the segment refer to the cell after the segment. Now, to solve this problem, we are looking at queries in reverse order: from last to first. This is how when we run a survey, all we have to do is paint exactly solut alakäsittelyssä $[l, r]$. Kaikki muut solut sisältävät jo lopullisen värinsä. Käytämme DSU:ta kaikkien tahraamattomien solujen iteraatioon. Löydämme vasemman maalaamattomimman solun segmentin sisältä, maalaamme sen uudelleen, ja osoittimilla siirrymme seuraavaan tyhjään soluun oikealle. Täällä voimme käyttää DSU: ta polun pakkaamisen kanssa, mutta emme voi käyttää liittoa sijoituksen / koon mukaan (koska on tärkeää, kuka tulee johtajaksi yhdistämisen jälkeen). Siksi monimutkaisuus $O (\log n)$ liitosta kohti (mikä on myös melko nopeaa). Toteutus: (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;= }= }= there= is= one= optimization:= we= can= use= union= by= rank,= if= we= store= the= next= unpainted= cell= in= an= additional= array= end[].= then= we= can= merge= two= sets= into= one= ranked= according= to= their= heuristics,= and= we= obtain= the= solution= in= $o(\alpha(n))$.= support= distances= up= to= representative= sometimes= in= specific= applications= of= the= dsu= you= need= to= maintain= the= distance= between= a= vertex= and= the= representative= of= its= set= (i.e.= the= path= length= in= the= tree= from= the= current= node= to= the= root= of= the= tree).= if= we= don't= use= path= compression ,= the= distance= is= just= the= number= of= recursive= calls.= but= this= will= be= inefficient.= however= it= is= possible= to= do= path= compression,= if= we= store= the= distance= to= the= parent= as= additional= information= for= each= node.= in= the= implementation= it= is= convenient= to= use= an= array= of= pairs= for= parent[]= and= the= function= find_set= now= returns= two= numbers:= the= representative= of= the= set,= and= the= distance= to= it.= void= make_set(int= v)= {= parent[v]=make_pair(v, 0);= rank[v]=0; } =>let= l= find_set(v).first; r= find_set(v).second; if (l != r) { if (rank[l] < rank[r]) swap(l, r); parent[r] = l; if (rank[l] == rank[r]) ++rank[l]; } } Applications and various improvements This section looks at multiple applications in the data structure, both trivial uses, and some improvements to the data structure. Components in the diagram This is one of the obvious applications of DSU. Formally, the problem is defined as follows: At first, we have an empty chart. We need to increase the top bonds and the directional edges. Edges. inquiries about the form (a, b) - are the top a and $$b in the same combined component chart? Here, we can directly apply the data structure and get a solution that handles the vert tip or edge and the addition of the survey within an average of almost standard time. This application is quite important because the krushal algorithm shows almost the same problem to find the minimum size tree. With DSU, we can improve $O(m \log n + n^2)$ $O(m \log n)$. Finding mapped components in an image One of DSU's applications is the following task: the image is $n \times m$ pixels. Originally, everyone is white, but then a few black pixels are drawn. You want to specify the size of each white merged portion of the final picture. To find a solution, simply iterate all the white pixels in the image, iterate iterated for each cell over its four neighbors, and if the neighbor is white, call union_sets. This way, we have a DSU with $n m$ knots that match the image pixels. The resulting trees in the DSU are the desired connected components. The problem can also be solved by DFS or BFS, but the method described here has an advantage: it can process the matrix line by line (i.e. deal with a row that we only need the previous and current row, and you only need one DSU built for the elements in the elements) $O(\min(n, m))$ in memory. Save more information about each series of DSU, which allows you to easily store additional information in sets. A simple example is the size of the sizes: the recording of sizes in the Union was already described according to the size section (the data was recorded by the current representative of the set). In the same way - by storing its representative nodes - you can also store other information about the set. Pack jumps along segment/Painting subarrays offline One common application of DSU is as follows: The vertex set is a set, and each vertex has an outbound edge at the other vertex. With DSU, you can find an end point that we can get to after following all the edges from the given starting point, almost in standard time. A good example of this application is the problem of painting subprocesses. We have a length segment L, each element initially has a color of 0. We need to paint a subrray with $[l, r]$ color c for each query (l, r, c). At the end, we want to find the ultimate color of each cell. We assume that we know all queries in advance, that is, the task is offline. For the solution, we can make a DSU, which for each cell stores a link to the next unblemished cell. Thus, at first, each cell points to itself. After you re-paint one of the requested segments, all cells in the segment refer to the cell after the segment. Now, to solve this problem, we are looking at queries in reverse order: from last to first. This is how when we run a survey, all we have to do is paint exactly solut alakäsittelyssä $[l, r]$. Kaikki muut solut sisältävät jo lopullisen värinsä. Käytämme DSU:ta kaikkien tahraamattomien solujen iteraatioon. Löydämme vasemman maalaamattomimman solun segmentin sisältä, maalaamme sen uudelleen, ja osoittimilla siirrymme seuraavaan tyhjään soluun oikealle. Täällä voimme käyttää DSU: ta polun pakkaamisen kanssa, mutta emme voi käyttää liittoa sijoituksen / koon mukaan (koska on tärkeää, kuka tulee johtajaksi yhdistämisen jälkeen). Siksi monimutkaisuus $O (\log n)$ liitosta kohti (mikä on myös melko nopeaa). Toteutus: (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;= }= }= there= is= one= optimization:= we= can= use= union= by= rank,= if= we= store= the= next= unpainted= cell= in= an= additional= array= end[].= then= we= can= merge= two= sets= into= one= ranked= according= to= their= heuristics,= and= we= obtain= the= solution= in= $o(\alpha(n))$.= support= distances= up= to= representative= sometimes= in= specific= applications= of= the= dsu= you= need= to= maintain= the= distance= between= a= vertex= and= the= representative= of= its= set= (i.e.= the= path= length= in= the= tree= from= the= current= node= to= the= root= of= the= tree).= if= we= don't= use= path= compression ,= the= distance= is= just= the= number= of= recursive= calls.= but= this= will= be= inefficient.= however= it= is= possible= to= do= path= compression,= if= we= store= the= distance= to= the= parent= as= additional= information= for= each= node.= in= the= implementation= it= is= convenient= to= use= an= array= of= pairs= for= parent[]= and= the= function= find_set= now= returns= two= numbers:= the= representative= of= the= set,= and= the= distance= to= it.= void= make_set(int= v)= {= parent[v]=make_pair(v, 0);= rank[v]=0; } =>let= l= find_set(v).first; r= find_set(v).second; if (l != r) { if (rank[l] < rank[r]) swap(l, r); parent[r] = l; if (rank[l] == rank[r]) ++rank[l]; } } Support the parity of the path length / Checking bipartiteness online In the same way as computing the path length to the leader, it is possible to maintain the parity of the length of the path before him. Why is this application in a separate paragraph? The unusual requirement of storing the parity of the path comes up in the following task: initially we are given an empty graph, it can be added edges, and we have to answer queries of the form is the connected component containing this vertex bipartite?. To solve this problem, we make a DSU for storing of the components and store the parity of the path up to the representative for each vertex. Thus we can quickly check if adding an edge leads to a violation of the bipartiteness or not: namely if the ends of the edge lie rank[b])= swap(a,= b);= parent[b]=make_pair(a, 1);= if= (rank[a]== rank[b])= rank[a]++;= }= }= support= the= parity= of= the= path= length= checking= bipartiteness= online= in= the= same= way= as= computing= the= path= length= to= the= leader,= it= is= possible= to= maintain= the= parity= of= the= length= of= the= path= before= him.= why= is= this= application= in= a= separate= paragraph?= the= unusual= requirement= of= storing= the= parity= of= the= path= comes= up= in= the= following= task:= initially= we= are= given= an= empty= graph,= it= can= be= added= edges,= and= we= have= to= answer= queries= of= the= form= is= the= connected= component= containing= this= vertex= bipartite?. = to= solve= this= problem,= we= make= a= dsu= for= storing= of= the= components= and= store= the= parity= of= the= path= up= to= the= representative= for= each= vertex.= thus= we= can= quickly= check= if= adding= an= edge= leads= to= a= violation= of= the= bipartiteness= or= not:= namely= if= the= ends= of= the= edge= lie=></ rank[b]) swap(a, b); parent[b] = make_pair(a, 1); if (rank[a] == rank[b]) rank[a]++; } } Support the parity of the path length / Checking bipartiteness online In the same way as computing the path length to the leader, it is possible to maintain the parity of the length of the path before him. Why is this application in a separate paragraph? The unusual requirement of storing the parity of the path comes up in the following task: initially we are given an empty graph, it can be added edges, and we have to answer queries of the form is the connected component containing this vertex bipartite?. To solve this problem, we make a DSU for storing of the components and store the parity of the path up to the representative for each vertex. Thus we can quickly check if adding an edge leads to a violation of the bipartiteness or not: namely if the ends of the edge lie > b) { a = find_set(a).first; b = find_set(b).first; if (a != b) { if (rank[a]</int,> </=> </=> sama yhdistetty komponentti ja sen pariteettipituus on sama kuin etuosalla, tämän reunan lisääminen tuottaa parittoman pitkän jakson ja komponentti menettää kaksiosaisuus-ominaisuuden. Ainoa vaikeutemme on laskea union_find. Jos lisäämme reunan (a, b) joka yhdistää kaksi liitettyä komponenttia yhdeksi, niin kun kiinnität puun toiseen, meidän on säädettävä pariteettia. Johdetään kaava, joka laskee setin johtajalle myönnetyn pariteettiarvon, joka liitetään toiseen sarjaan. Anna x olla polun pituuden pariteetti vertexistä $a $ johtajaansa $A $ ja $y $ polun pituuden pariteettina kärkipisteistä $b $ sen johtajaan $B $ ja $t $ haluttuun pariteettiin, joka meidän on määritettävä $B $ yhdistämisen jälkeen. Polku sisältää kolme osaa: B - $b $, $b $ - $a $, joka on yhdistetty yhdellä reunalla ja jolla on siksi pariteetti 1, ja a - A. Siksi saamme kaavan ($\oplus$ tarkoittaa XOR-toimintoa): $$t = x \oplus y \oplus 1$$ Näin riippumatta siitä, kuinka monta liitosta suoritamme, reunojen pariteetti kuljetetaan johtajasta toiseen. Annamme DSU:n täytäntöönpanon, joka tukee yhdenvertaisuutta. Kuten edellisessä osassa, käytämme paria esivansorin ja pariteettien tallentamiseen. Lisäksi säilytämme matriisissa kaksipuoluesivustoa[] riippumatta siitä, make_pair onko se edelleen kaksiosainen vai ei make_set. 0); rank[v] = 0; bipartiitti[v] = tosi; } pari<int, int=> find_set(int v) { jos (v != vanhempi[v].ensimmäinen) { int pariteetti = parent[v].second; parent[v] = find_set(parent[v].first); parent[v].second ^= pariteetti; } palautusvanhempi[v]; } void add_edge(int a, int b) {<int, int=> pair 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]; } Offline RMQ (range minimum query) in $O(\alpha(n))$ on average / Arpa's trick We are given an array a[] and we have to compute some minima in given segments of the array. The idea to solve this problem with DSU is the following: We will iterate over the array and when we are at the ith element we will answer all queries (L, R) with R == i. To do this efficiently we will keep a DSU using the first i elements with the following structure: the parent of an element is the next smaller element to the right of it. Then using this structure the answer to a query will be the a[find_set(L)], the smallest number to the right of L. This approach obviously only works offline, i.e. if we know all queries beforehand. It is easy to see that we can apply path compression. And we can also use Union by rank, if we store the actual leader rank[b])= swap= (a,= b);= parent[b]=make_pair(a, bipartite[a]= &=bipartite[b]; if= (rank[a]== rank[b])= ++rank[a];= }= }= bool= is_bipartite(int= v)= {= return= bipartite[find_set(v).first];= }= offline= rmq= (range= minimum= query)= in= $o(\alpha(n))$= on= average= arpa's= trick= we= are= given= an= array= a[]= and= we= have= to= compute= some= minima= in= given= segments= of= the= array.= the= idea= to= solve= this= problem= with= dsu= is= the= following:= we= will= iterate= over= the= array= and= when= we= are= at= the= ith= element= we= will= answer= all= queries= (l,= r)= with= r== i.= to= do= this= efficiently= we= will= keep= a= dsu= using= the= first= i= elements= with= the= following= structure:= the= parent= of= an= element= is= the= next= smaller= element= to= the= right= of= it.= then= using= this= structure= the= answer= to= a= query= will= be= the= a[find_set(l)] ,= the= smallest= number= to= the= right= of= l.= this= approach= obviously= only= works= offline,= i.e.= if= we= know= all= queries= beforehand.= it= is= easy= to= see= that= we= can= apply= path= compression.= and= we= can= also= use= union= by= rank,= if= we= store= the= actual= leader=></ 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]; } Offline RMQ (range minimum query) in $O(\alpha(n))$ on average / Arpa's trick We are given an array a[] and we have to compute some minima in given segments of the array. The idea to solve this problem with DSU is the following: We will iterate over the array and when we are at the ith element we will answer all queries (L, R) with R == i. To do this efficiently we will keep a DSU using the first i elements with the following structure: the parent of an element is the next smaller element to the right of it. Then using this structure the answer to a query will be the a[find_set(L)], the smallest number to the right of L. This approach obviously only works offline, i.e. if we know all queries beforehand. It is easy to see that we can apply path compression. And we can also use Union by rank, if we store the actual leader > </int,> </int,> </int,> </int,> a separate matrix. struct Query { int L, R, idx; }; <int>vector response; <> <Query>vector > tank; the container[i] contains all queries with 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)]; } Today, this algorithm is known as arpa's trick. It is named after AmirReza Poorakhavan, who independently discovered and popularized this technique. Although this algorithm existed even before he was found. Offline LCA (the smallest common ancestor of the tree) $O(\alpha(n)$ average Algorithm to find LCA is discussed in the article The smallest common ancestor - Tarjan's offline algorithm. This algorithm compares favorably with other algorithms to find LCA because of its simplicity (especially when compared to an optimal algorithm such as that of Farach-Colton and Bender). Storing DSU in an explicitly placed list/Applications of this idea when combining different data structures One of the alternative ways to save DSU is to keep each batch in the form of an explicitly stored list of its elements. At the same time, each element also stores a reference to the set representative. At first glance, this looks like an inefficient data structure: by combining two sets, we need to add one list to the end of the other and update leadership in one list. However, it turned out that weighting carcover (similar to Union by size) can significantly reduce asymptomatic complexity: $O(m + n \log n)$$m$ to conduct queries about $n$elements. Emphasis heuristically means that we always add the smaller one from two series to the larger series. Adding one set to another is easy to implement union_sets time in relation to the size of the added set. And looking for a find_set takes $O(1)$ with this storage method. Let us prove time $O(m + n \log n)$ m to complete queries. We correct the arbitrary x1 and calculate how often it was touched union_sets. When x is touched the first time, the size of the new set is at least $2. When touched the second time, the resulting set size is at least $4, as the smaller set is added to the larger one. And so on. This means that $$x can only be transferred in the first $\log n$ mail merge. Thus, the sum of all the top $O (n \log n)$ plus $O(1)$ for each request. Here is the implementation:<int> vector lst[MAXN]; int parent[MAXN]; void make_set(int v) { lst[v] =<int>vector (1, v); parent[v] = v; } int find_set(int v) { return parent[v]; } voided union_sets(int a, int b) { a = find_set(a); b = < lst[b].size()) swap(a, b); lst[b].size())= swap(a,= lst[b].size()) swap(a, b); > find_set(b); jos (a != b) { jos (lst[a].size()</int> </int> </int> </int> </Query> </int> </int> (!lst[b].blank()) { int v = lst[b].back(); lst[b].pop_back(); parent[v] = a; lst[a].push_back (v); } } Adding a smaller part of this idea to a larger part can also be used with many solutions that have nothing to do with DSU. For example, take into account the following problem: we are given a tree, each magazine has a number assigned (the same number can appear several times in different magazines). We want to count the number of different numbers in the sub tree for each node in the tree. By applying the same idea to this task, it is possible to find this solution: we can introduce DFS, which uses the pointer to return a set of integers - in the lower tree of the number list. Then to get a response to the current node (unless of course it's a magazine), we call DFS for all the kids in this knot and combine all the series received together. The resulting set size is the answer to the current node. If you want to effectively combine multiple sets, we only use the recipe described above: we combine the kits simply by adding smaller ones to larger ones. Eventually, $O (n \log^2 n)$ solution because one number is added to the series no more than $O(\log n)$ times. How to maintain DSU by maintaining a clear wooden structure/finding an online bridge $O(\alpha(n)$) for an average One of DSU's most powerful applications is that it allows you to save both as compressed and unpackaged wood. A compressed form can be used to connect trees and check it if there are two vertexes in the same tree and an uncompressed shape can be used - for example - to search for paths between two given vertexes or other paths to the tree structure. In implementation, this means that in addition to the parent of the packed ancestral table[], we must keep the table of unpackaged ancestors real_parent[]. It is trivial that maintaining this additional table does not exacerbate complexity: changes only happen when we combine two trees, and in only one element. On the other hand, when applied in practice, we often have to connect the trees using a certain edge that uses two root nodes. This means that we have no choice but to root one of the trees again (to make the ends of the edge a new root of the tree). At first glance, it seems that this re-rooting is very expensive and greatly exacerbates the complexity of time. In fact, to root the tree in vertex v1, we need to move from the vertex to the old root and change direction to the top and real_parent[] all the nodes on the path. In reality, however, it is not so bad, we can only root the smaller of two trees that resemble the ideas of the previous parts, and get an average of $O (\log n)$. For more information (including proof of the complexity of time), see Find bridges online. Historical retrospective Data structure has been known for a long time. This method is a way to make this structure in the form in the form Galler and Fisher apparently first described the forest of trees in 1964 (Galler, Fisher, Improved Equivalence Algorithm), but a full analysis of the complexity of time was done much later. McIlroy and Morris have developed an optimization path according to compression and union order, and regardless of that, Tritter as well. Hopcroft and Ullman demonstrated the complexity of time in 1973 $O(\log^\star n)$ (Hopcroft, Ullman Set-merging algorithms) - here $\log^\star$ is an iterated logaritmis (this is a slow-growing function, but still not as slow as the inverted Ackermann function). The first $O(\alpha(n)$ was shown in 1975 (the effectiveness of Tarja's Good But Not Linear Union Algorithm). Later, in 1985, he and Leeuwen published several complexity analyses of a variety of value heurisms and ways to sum up the path (Tarjan, Leeuwe's Worst Case Analysis set union algorithms). Finally, in 1989, Fredman and Sachs proved that in an approved calculation model, any algorithm that concerns a fragmented union problem must work on average at least $O(\alpha(n)$ in time (Fredman, Saks, the complexity of the cell sensor of dynamic data structures). However, it should also be noted that there are several articles that dispute this initial appreciation and claim that DSU, which has a path compression and union by investment, runs on average $O(1)$ in time (Zhang Union-Find Problem Is Linear, Wu, Otoo Simpler proof of the average case complexity of union-find with Path Compression). Problems