Last update: May 6, 2023 Translated From: e-maxx.ru

Disjoint Set Union

This article discusses the data structure **Disjoint Set Union** or **DSU**. Often it is also called **Union Find** because of its two main operations.

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, it can create a set from a new element.

Thus the basic interface of this data structure consists of only three operations:

- `make_set(v)` - creates a new set consisting of the new element `v`
- `union_sets(a, b)` - merges the two specified sets (the set in which the element `a` is located, and the set in which the element `b` is located)
- `find_set(v)` - returns the representative (also called leader) of the set that contains the element `v`. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after `union_sets` calls). This representative can be used to check if two elements are part of the same set or not. `a` and `b` are exactly in the same set, if `find_set(a) == find_set(b)`. Otherwise they are in different sets.

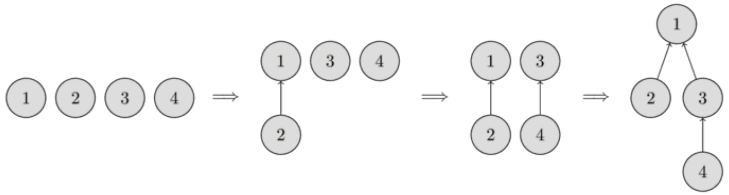
As described in more detail later, the data structure allows you to do each of these operations in almost $O(1)$ time on average.

Also in one of the subsections an alternative structure of a DSU is explained, which achieves a slower average complexity of $O(\log n)$, but can be more powerful than the regular DSU structure.

Build an efficient data structure

We will store the sets in the form of **trees**: each tree will correspond to one set. And the root of the tree will be the representative/leader of the set.

In the following image you can see the representation of such trees.



In the beginning, every element starts as a single set, therefore each vertex is its own tree. Then we combine the set containing the element 1 and the set containing the element 2. Then we combine the set containing the element 3 and the set containing the element 4. And in the last step, we combine the set containing the element 1 and the set containing the element 3.

For the implementation this means that we will have to maintain an array `parent` that stores a reference to its immediate ancestor in the tree.

Naive implementation

We can already write the first implementation of the Disjoint Set Union data structure. It will be pretty inefficient at first, but later we can improve it using two optimizations, so that it will take nearly constant time for each function call.

As we said, all the information about the sets of elements will be kept in an array `parent`.

To create a new set (operation `make_set(v)`), we simply create a tree with root in the vertex `v`, meaning that it is its own ancestor.

To combine two sets (operation `union_sets(a, b)`), we first find the representative of the set in which `a` is located, and the representative of the set in which `b` is located. If the representatives are identical, that we have nothing to do, the sets are already merged. Otherwise, we can simply specify that one of the representatives is the parent of the other representative - thereby combining the two trees.

Finally the implementation of the find representative function (operation `find_set(v)`): we simply climb the ancestors of the vertex `v` until we reach the root, i.e. a vertex such that the reference to the ancestor leads to itself. This operation is easily implemented 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)
        parent[b] = a;
}
```

However this implementation is inefficient. It is easy to construct an example, so that the trees degenerate into long chains. In that case each call `find_set(v)` can take $O(n)$ time.

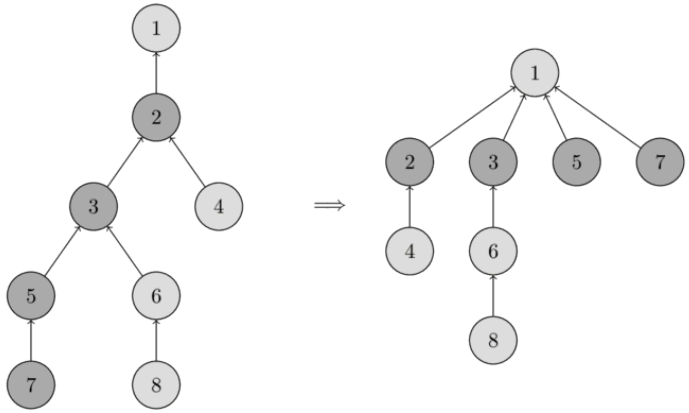
This is far away from the complexity that we want to have (nearly constant time). Therefore we will consider two optimizations that will allow to significantly accelerate the work.

Path compression optimization

This optimization is designed for speeding up `find_set`.

If we call `find_set(v)` for some vertex `v`, we actually find the representative `p` for all vertices that we visit on the path between `v` and the actual representative `p`. The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to `p`.

You can see the operation in the following image. On the left there is a tree, and on the right side there is the compressed tree after calling `find_set(7)`, which shortens the paths for the visited nodes 7, 5, 3 and 2.



The new implementation of `find_set` is as follows:


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

The simple implementation does what was intended: first find the representative of the set (root vertex), and then in the process of stack unwinding the visited nodes are attached directly to the representative.

This simple modification of the operation already achieves the time complexity $O(\log n)$ per call on average (here without proof). There is a second modification, that will make it even faster.

Union by size / rank

In this optimization we will change the `union_set` operation. To be precise, we will change which tree gets attached to the other one. In the naive implementation the second tree always got attached to the first one. In practice that can lead to trees containing chains of length $O(n)$. With this optimization we will avoid this by choosing very carefully which tree gets attached.

There are many possible heuristics that can be used. Most popular are the following two approaches: In the first approach we use the size of the trees as rank, and in the second one we use the depth of the tree (more precisely, the upper bound on the tree depth, because the depth will get smaller when applying path compression).

In both approaches the essence of the optimization is the same: we attach the tree with the lower rank to the one with the bigger rank.

Here is the implementation of union by size:

```
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];
    }
}
```

And here is the implementation of union by rank based on the depth of the trees:

```
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]++;
    }
}
```

Both optimizations are equivalent in terms of time and space complexity. So in practice you can use any of them.

Time complexity

As mentioned before, if we combine both optimizations - path compression with union by size / rank - we will reach nearly constant time queries. It turns out, that the final amortized time complexity is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very slowly. In fact it grows so slowly, that it doesn't exceed 4 for all reasonable n (approximately $n < 10^{600}$).

Amortized complexity is the total time per operation, evaluated over a sequence of multiple operations. The idea is to guarantee the total time of the entire sequence, while allowing single operations to be much slower then the amortized time. E.g. in our case a single call might take $O(\log n)$ in the worst case, but if we do m such calls back to back we will end up with an average time of $O(\alpha(n))$.

We will also not present a proof for this time complexity, since it is quite long and complicated.

Also, it's worth mentioning that DSU with union by size / rank, but without path compression works in $O(\log n)$ time per query.

Linking by index / coin-flip linking

Both union by rank and union by size require that you store additional data for each set, and maintain these values during each union operation. There exist also a randomized algorithm, that simplifies the union operation a little bit: linking by index.

We assign each set a random value called the index, and we attach the set with the smaller index to the one with the larger one. It is likely that a bigger set will have a bigger index than the smaller set, therefore this operation is closely related to union by size. In fact it can be proven, that this operation has the same time complexity as union by size. However in practice it is slightly slower than union by size.

You can find a proof of the complexity and even more union techniques here.


```
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's a common misconception that just flipping a coin, to decide which set we attach to the other, has the same complexity. However that's not true. The paper linked above conjectures that coin-flip linking combined with path compression has complexity $\Omega\left(n \frac{\log n}{\log \log n}\right)$. And in benchmarks it performs a lot worse than union by size/rank or linking by index.

```
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

In this section we consider several applications of the data structure, both the trivial uses and some improvements to the data structure.

Connected components in a graph

This is one of the obvious applications of DSU.

Formally the problem is defined in the following way: Initially we have an empty graph. We have to add vertices and undirected edges, and answer queries of the form (a, b) - "are the vertices a and b in the same connected component of the graph?"

Here we can directly apply the data structure, and get a solution that handles an addition of a vertex or an edge and a query in nearly constant time on average.

This application is quite important, because nearly the same problem appears in Kruskal's algorithm for finding a minimum spanning tree. Using DSU we can improve the $O(m \log n + n^2)$ complexity to $O(m \log n)$.

Search for connected components in an image

One of the applications of DSU is the following task: there is an image of $n \times m$ pixels. Originally all are white, but then a few black pixels are drawn. You want to determine the size of each white connected component in the final image.

For the solution we simply iterate over all white pixels in the image, for each cell iterate over its four neighbors, and if the neighbor is white call `union_sets`. Thus we will have a DSU with nm nodes corresponding to 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 row by row (i.e. to process a row we only need the previous and the current row, and only need a DSU built for the elements of one row) in $O(\min(n, m))$ memory.

Store additional information for each set

DSU allows you to easily store additional information in the sets.

A simple example is the size of the sets: storing the sizes was already described in the Union by size section (the information was stored by the current representative of the set).

In the same way - by storing it at the representative nodes - you can also store any other information about the sets.

Compress jumps along a segment / Painting subarrays offline

One common application of the DSU is the following: There is a set of vertices, and each vertex has an outgoing edge to another vertex. With DSU you can find the end point, to which we get after following all edges from a given starting point, in almost constant time.

A good example of this application is the **problem of painting subarrays**. We have a segment of length L, each element initially has the color 0. We have to repaint the subarray $[l, r]$ with the color c for each query (l, r, c). At the end we want to find the final color of each cell. We assume that we know all the queries in advance, i.e. the task is offline.

For the solution we can make a DSU, which for each cell stores a link to the next unpainted cell. Thus initially each cell points to itself. After painting one requested repaint of a segment, all cells from that segment will point to the cell after the segment.

Now to solve this problem, we consider the queries **in the reverse order**: from last to first. This way when we execute a query, we only have to paint exactly the unpainted cells in the subarray $[l, r]$. All other cells

already contain their final color. To quickly iterate over all unpainted cells, we use the DSU. We find the left-most unpainted cell inside of a segment, repaint it, and with the pointer we move to the next empty cell to the right.

Here we can use the DSU with path compression, but we cannot use union by rank / size (because it is important who becomes the leader after the merge). Therefore the complexity will be $O(\log n)$ per union (which is also quite fast).

Implementation:

```
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;
    }
}
```

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;
}

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]++;
    }
}
```

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 in the same connected component and have the same parity length to the leader, then adding this edge will produce a cycle of odd length, and the component will lose the bipartiteness property.

The only difficulty that we face is to compute the parity in the `union_find` method.

If we add an edge (a, b) that connects two connected components into one, then when you attach one tree to another we need to adjust the parity.

Let's derive a formula, which computes the parity issued to the leader of the set that will get attached to another set. Let x be the parity of the path length from vertex a up to its leader A, and y as the parity of the path length from vertex b up to its leader B, and t the desired parity that we have to assign to B after the merge. The path contains the of the three parts: from B to b, from b to a, which is connected by one edge and therefore has parity 1, and from a to A. Therefore we receive the formula ($\oplus$ denotes the XOR operation):

$$t = x \oplus y \oplus 1$$

Thus regardless of how many joins we perform, the parity of the edges is carried from one leader to another.

We give the implementation of the DSU that supports parity. As in the previous section we use a pair to store the ancestor and the parity. In addition for each set we store in the array `bipartite[]` whether it is still bipartite or not.

```
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];
}
```

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 `i` th 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 in an separate array.

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

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

`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)];
    }
}
```

Nowadays 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 already before his discovery.

Offline LCA (lowest common ancestor in a tree) in $O(\alpha(n))$ on average

The algorithm for finding the LCA is discussed in the article Lowest Common Ancestor - Tarjan's off-line algorithm. This algorithm compares favorable with other algorithms for finding the LCA due to its simplicity (especially compared to an optimal algorithm like the one from Farach-Colton and Bender).

Storing the DSU explicitly in a set list / Applications of this idea when merging various data structures

One of the alternative ways of storing the DSU is the preservation of each set in the form of an **explicitly stored list of its elements**. At the same time each element also stores the reference to the representative

of his set.

At first glance this looks like an inefficient data structure: by combining two sets we will have to add one list to the end of another and have to update the leadership in all elements of one of the lists.

However it turns out, the use of a **weighting heuristic** (similar to Union by size) can significantly reduce the asymptotic complexity: $O(m + n \log n)$ to perform m queries on the n elements.

Under weighting heuristic we mean, that we will always **add the smaller of the two sets to the bigger set**. Adding one set to another is easy to implement in `union_sets` and will take time proportional to the size of the added set. And the search for the leader in `find_set` will take $O(1)$ with this method of storing.

Let us prove the **time complexity** $O(m + n \log n)$ for the execution of m queries. We will fix an arbitrary element x and count how often it was touched in the merge operation `union_sets`. When the element x gets touched the first time, the size of the new set will be at least 2. When it gets touched the second time, the resulting set will have size of at least 4, because the smaller set gets added to the bigger one. And so on. This means, that x can only be moved in at most $\log n$ merge operations. Thus the sum over all vertices gives $O(n \log n)$ plus $O(1)$ for each request.

Here is an implementation:

```
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);
        }
    }
}
```

This idea of adding the smaller part to a bigger part can also be used in a lot of solutions that have nothing to do with DSU.

For example consider the following **problem**: we are given a tree, each leaf has a number assigned (same number can appear multiple times on different leaves). We want to compute the number of different numbers in the subtree for every node of the tree.

Applying to this task the same idea it is possible to obtain this solution: we can implement a DFS, which will return a pointer to a set of integers - the list of numbers in that subtree. Then to get the answer for the current node (unless of course it is a leaf), we call DFS for all children of that node, and merge all the received sets together. The size of the resulting set will be the answer for the current node. To efficiently combine multiple sets we just apply the above-described recipe: we merge the sets by simply adding smaller ones to larger. In the end we get a $O(n \log^2 n)$ solution, because one number will only added to a set at most $O(\log n)$ times.

Storing the DSU by maintaining a clear tree structure / Online bridge finding in $O(\alpha(n))$ on average

One of the most powerful applications of DSU is that it allows you to store both as **compressed and uncompressed trees**. The compressed form can be used for merging of trees and for the verification if two vertices are in the same tree, and the uncompressed form can be used - for example - to search for paths between two given vertices, or other traversals of the tree structure.

In the implementation this means that in addition to the compressed ancestor array `parent[]` we will need to keep the array of uncompressed ancestors `real_parent[]`. It is trivial that maintaining this additional array will not worsen the complexity: changes in it only occur when we merge two trees, and only in one element.

On the other hand when applied in practice, we often need to connect trees using a specified edge other that using the two root nodes. This means that we have no other choice but to re-root one of the trees (make the ends of the edge the new root of the tree).

At first glance it seems that this re-rooting is very costly and will greatly worsen the time complexity. Indeed, for rooting a tree at vertex v we must go from the vertex to the old root and change directions in `parent[]` and `real_parent[]` for all nodes on that path.

However in reality it isn't so bad, we can just re-root the smaller of the two trees similar to the ideas in the previous sections, and get $O(\log n)$ on average.

More details (including proof of the time complexity) can be found in the article Finding Bridges Online.

Historical retrospective

The data structure DSU has been known for a long time.

This way of storing this structure in the form **of a forest of trees** was apparently first described by Galler and Fisher in 1964 (Galler, Fisher, "An Improved Equivalence Algorithm), however the complete analysis of

Last update: June 8, 2022 Translated From: e-maxx.ru

Depth First Search

Depth First Search is one of the main graph algorithms.

Depth First Search finds the lexicographical first path in the graph from a source vertex u to each vertex. Depth First Search will also find the shortest paths in a tree (because there only exists one simple path), but on general graphs this is not the case.

The algorithm works in $O(m+n)$ time where n is the number of vertices and m is the number of edges.

Description of the algorithm

The idea behind DFS is to go as deep into the graph as possible, and backtrack once you are at a vertex without any unvisited adjacent vertices.

It is very easy to describe / implement the algorithm recursively: We start the search at one vertex. After visiting a vertex, we further perform a DFS for each adjacent vertex that we haven't visited before. This way we visit all vertices that are reachable from the starting vertex.

For more details check out the implementation.

Applications of Depth First Search

- Find any path in the graph from source vertex u to all vertices.
- Find lexicographical first path in the graph from source u to all vertices.
- Check if a vertex in a tree is an ancestor of some other vertex:

  At the beginning and end of each search call we remember the entry and exit "time" of each vertex. Now you can find the answer for any pair of vertices (i, j) in $O(1)$: vertex i is an ancestor of vertex j if and only if $\text{entry}[i] < \text{entry}[j]$ and $\text{exit}[i] > \text{exit}[j]$.
- Find the lowest common ancestor (LCA) of two vertices.
- Topological sorting:

  Run a series of depth first searches so as to visit each vertex exactly once in $O(n+m)$ time. The required topological ordering will be the vertices sorted in descending order of exit time.
- Check whether a given graph is acyclic and find cycles in a graph. (As mentioned above by counting back edges in every connected components).
- Find strongly connected components in a directed graph:

  First do a topological sorting of the graph. Then transpose the graph and run another series of depth first searches in the order defined by the topological sort. For each DFS call the component created by it is a strongly connected component.
- Find bridges in an undirected graph:

  First convert the given graph into a directed graph by running a series of depth first searches and making each edge directed as we go through it, in the direction we went. Second, find the strongly connected components in this directed graph. Bridges are the edges whose ends belong to different strongly connected components.

Classification of edges of a graph

We can classify the edges using the entry and exit time of the end nodes u and v of the edges (u, v). These classifications are often used for problems like finding bridges and finding articulation points.

We perform a DFS and classify the encountered edges using the following rules:

If v is not visited:

- Tree Edge - If v is visited after u then edge (u, v) is called a tree edge. In other words, if v is visited for the first time and u is currently being visited then (u, v) is called tree edge. These edges form a DFS tree and hence the name tree edges.

If v is visited before u:

- Back edges - If v is an ancestor of u, then the edge (u, v) is a back edge. v is an ancestor exactly if we already entered v, but not exited it yet. Back edges complete a cycle as there is a path from ancestor v to descendant u (in the recursion of DFS) and an edge from descendant u to ancestor v (back edge), thus a cycle is formed. Cycles can be detected using back edges.
- Forward Edges - If v is a descendant of u, then edge (u, v) is a forward edge. In other words, if we already visited and exited v and $\text{entry}[u] < \text{entry}[v]$ then the edge (u, v) forms a forward edge.

- Cross Edges: if v is neither an ancestor or descendant of u, then edge (u, v) is a cross edge. In other words, if we already visited and exited v and $\text{entry}[u] > \text{entry}[v]$ then (u, v) is a cross edge.

Note: Forward edges and cross edges only exist in directed graphs.

Implementation

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<bool> visited;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
}
```

This is the most simple implementation of Depth First Search. As described in the applications it might be useful to also compute the entry and exit times and vertex color. We will color all vertices with the color 0, if we haven't visited them, with the color 1 if we visited them, and with the color 2, if we already exited the vertex.

Here is a generic implementation that additionally computes those:

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<int> color;

vector<int> time_in, time_out;
int dfs_timer = 0;

void dfs(int v) {
    time_in[v] = dfs_timer++;
    color[v] = 1;
    for (int u : adj[v])
        if (color[u] == 0)
            dfs(u);
    color[v] = 2;
    time_out[v] = dfs_timer++;
}
```

Last update: January 18, 2023 Translated From: e-maxx.ru

Breadth-first search

Breadth first search is one of the basic and essential searching algorithms on graphs.

As a result of how the algorithm works, the path found by breadth first search to any node is the shortest path to that node, i.e the path that contains the smallest number of edges in unweighted graphs.

The algorithm works in $O(n + m)$ time, where n is number of vertices and m is the number of edges.

Description of the algorithm

The algorithm takes as input an unweighted graph and the id of the source vertex s. The input graph can be directed or undirected, it does not matter to the algorithm.

The algorithm can be understood as a fire spreading on the graph: at the zeroth step only the source s is on fire. At each step, the fire burning at each vertex spreads to all of its neighbors. In one iteration of the algorithm, the "ring of fire" is expanded in width by one unit (hence the name of the algorithm).

More precisely, the algorithm can be stated as follows: Create a queue q which will contain the vertices to be processed and a Boolean array $used[]$ which indicates for each vertex, if it has been lit (or visited) or not.

Initially, push the source s to the queue and set $used[s] = true$, and for all other vertices v set $used[v] = false$. Then, loop until the queue is empty and in each iteration, pop a vertex from the front of the queue. Iterate through all the edges going out of this vertex and if some of these edges go to vertices that are not already lit, set them on fire and place them in the queue.

As a result, when the queue is empty, the "ring of fire" contains all vertices reachable from the source s, with each vertex reached in the shortest possible way. You can also calculate the lengths of the shortest paths (which just requires maintaining an array of path lengths $d[]$) as well as save information to restore all of these shortest paths (for this, it is necessary to maintain an array of "parents" $p[]$, which stores for each vertex the vertex from which we reached it).

Implementation

We write code for the described algorithm in C++ and Java.

C++

```
vector<vector<int>> adj;  // adjacency list representation
int n; // number of nodes
int s; // source vertex

queue<int> q;
vector<bool> used(n);
vector<int> d(n), p(n);

q.push(s);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

Java

```
ArrayList<ArrayList<Integer>> adj = new ArrayList<>(); // adjacency list representation

int n; // number of nodes
int s; // source vertex


LinkedList<Integer> q = new LinkedList<Integer>();
boolean used[] = new boolean[n];
int d[] = new int[n];
int p[] = new int[n];

q.push(s);
used[s] = true;
p[s] = -1;
while (!q.isEmpty()) {
    int v = q.pop();
    for (int u : adj.get(v)) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

If we have to restore and display the shortest path from the source to some vertex u, it can be done in the following manner:

C++

```
if (!used[u]) {
    cout << "No path!";
} else {
    vector<int> path;
    for (int v = u; v != -1; v = p[v])
        path.push_back(v);
    reverse(path.begin(), path.end());
```



```
    cout << "Path: ";
    for (int v : path)
        cout << v << " ";
}
```

Java

```
if (!used[u]) {
    System.out.println("No path!");
} else {
    ArrayList<Integer> path = new ArrayList<Integer>();
    for (int v = u; v != -1; v = p[v])
        path.add(v);
    Collections.reverse(path);
    for(int v : path)
        System.out.println(v);
}
```

Applications of BFS

- Find the shortest path from a source to other vertices in an unweighted graph.
- Find all connected components in an undirected graph in $O(n + m)$ time: To do this, we just run BFS starting from each vertex, except for vertices which have already been visited from previous runs. Thus, we perform normal BFS from each of the vertices, but do not reset the array $used[]$ each and every time we get a new connected component, and the total running time will still be $O(n + m)$ (performing multiple BFS on the graph without zeroing the array $used[]$ is called a series of breadth first searches).
- Finding a solution to a problem or a game with the least number of moves, if each state of the game can be represented by a vertex of the graph, and the transitions from one state to the other are the edges of the graph.
- Finding the shortest path in a graph with weights 0 or 1: This requires just a little modification to normal breadth-first search: Instead of maintaining array $used[]$, we will now check if the distance to vertex is shorter than current found distance, then if the current edge is of zero weight, we add it to the front of the queue else we add it to the back of the queue.This modification is explained in more detail in the article 0-1 BFS.
- Finding the shortest cycle in a directed unweighted graph: Start a breadth-first search from each vertex. As soon as we try to go from the current vertex back to the source vertex, we have found the shortest cycle containing the source vertex. At this point we can stop the BFS, and start a new BFS from the next vertex. From all such cycles (at most one from each BFS) choose the shortest.
- Find all the edges that lie on any shortest path between a given pair of vertices (a, b). To do this, run two breadth first searches: one from a and one from b. Let $d_a[]$ be the array containing shortest distances obtained from the first BFS (from a) and $d_b[]$ be the array containing shortest distances obtained from the second BFS from b. Now for every edge (u, v) it is easy to check whether that edge lies on any shortest path between a and b: the criterion is the condition $d_a[u] + 1 + d_b[v] = d_a[b]$.
- Find all the vertices on any shortest path between a given pair of vertices (a, b). To accomplish that, run two breadth first searches: one from a and one from b. Let $d_a[]$ be the array containing shortest distances obtained from the first BFS (from a) and $d_b[]$ be the array containing shortest distances obtained from the second BFS (from b). Now for each vertex it is easy to check whether it lies on any shortest path between a and b: the criterion is the condition $d_a[v] + d_b[v] = d_a[b]$.

Last update: September 17, 2023 Translated From: e-maxx.ru

Maximum flow - Ford-Fulkerson and Edmonds-Karp

The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method for computing a maximal flow in a flow network.

Flow network

First let's define what a **flow network**, a **flow**, and a **maximum flow** is.

A **network** is a directed graph G with vertices V and edges E combined with a function c, which assigns each edge $e \in E$ a non-negative integer value, the **capacity** of e. Such a network is called a **flow network**, if we additionally label two vertices, one as **source** and one as **sink**.

A **flow** in a flow network is function f, that again assigns each edge e a non-negative integer value, namely the flow. The function has to fulfill the following two conditions:

The flow of an edge cannot exceed the capacity.

$$f(e) \leq c(e)$$

And the sum of the incoming flow of a vertex u has to be equal to the sum of the outgoing flow of u except in the source and sink vertices.

$$\sum_{(v,u) \in E} f((v,u)) = \sum_{(u,v) \in E} f((u,v))$$

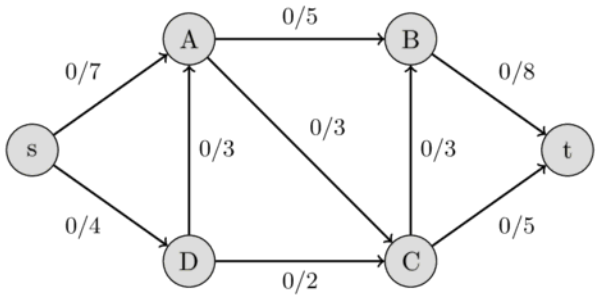
The source vertex s only has an outgoing flow, and the sink vertex t has only incoming flow.

It is easy to see that the following equation holds:

$$\sum_{(s,u) \in E} f((s,u)) = \sum_{(u,t) \in E} f((u,t))$$

A good analogy for a flow network is the following visualization: We represent edges as water pipes, the capacity of an edge is the maximal amount of water that can flow through the pipe per second, and the flow of an edge is the amount of water that currently flows through the pipe per second. This motivates the first flow condition. There cannot flow more water through a pipe than its capacity. The vertices act as junctions, where water comes out of some pipes, and then, these vertices distribute the water in some way to other pipes. This also motivates the second flow condition. All the incoming water has to be distributed to the other pipes in each junction. It cannot magically disappear or appear. The source s is origin of all the water, and the water can only drain in the sink t.

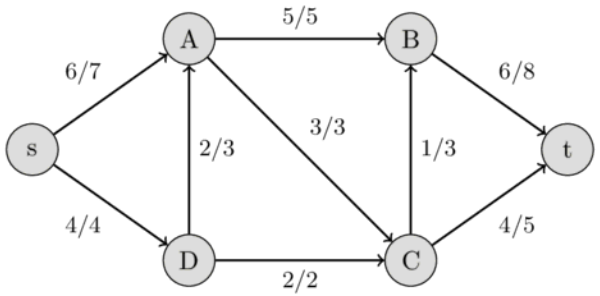
The following image shows a flow network. The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.



The value of the flow of a network is the sum of all the flows that get produced in the source s, or equivalently to the sum of all the flows that are consumed by the sink t. A **maximal flow** is a flow with the maximal possible value. Finding this maximal flow of a flow network is the problem that we want to solve.

In the visualization with water pipes, the problem can be formulated in the following way: how much water can we push through the pipes from the source to the sink?

The following image shows the maximal flow in the flow network.



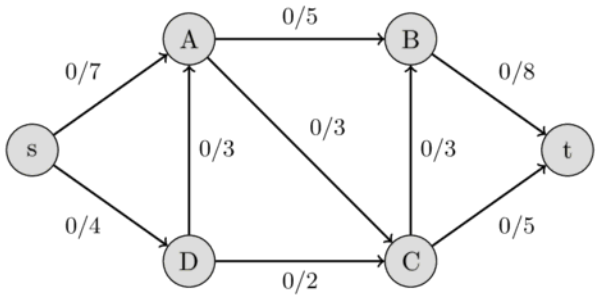
Ford-Fulkerson method

Let's define one more thing. A **residual capacity** of a directed edge is the capacity minus the flow. It should be noted that if there is a flow along some directed edge (u, v), then the reversed edge has capacity 0 and we can define the flow of it as $f((v, u)) = -f((u, v))$. This also defines the residual capacity for all the reversed edges. We can create a **residual network** from all these edges, which is just a network with the same vertices and edges, but we use the residual capacities as capacities.

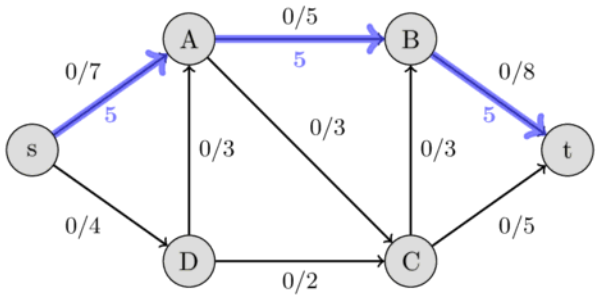
The Ford-Fulkerson method works as follows. First, we set the flow of each edge to zero. Then we look for an **augmenting path** from s to t. An augmenting path is a simple path in the residual graph, i.e. along the edges whose residual capacity is positive. If such a path is found, then we can increase the flow along these edges. We keep on searching for augmenting paths and increasing the flow. Once an augmenting path doesn't exist anymore, the flow is maximal.

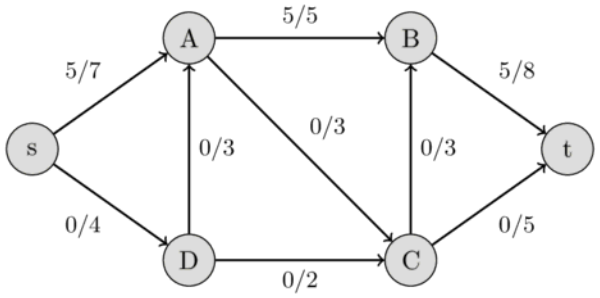
Let us specify in more detail, what increasing the flow along an augmenting path means. Let C be the smallest residual capacity of the edges in the path. Then we increase the flow in the following way: we update $f((u, v)) \mathrel{+}= C$ and $f((v, u)) \mathrel{-}= C$ for every edge (u, v) in the path.

Here is an example to demonstrate the method. We use the same flow network as above. Initially we start with a flow of 0.

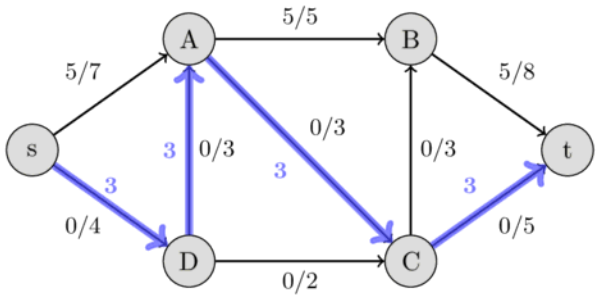


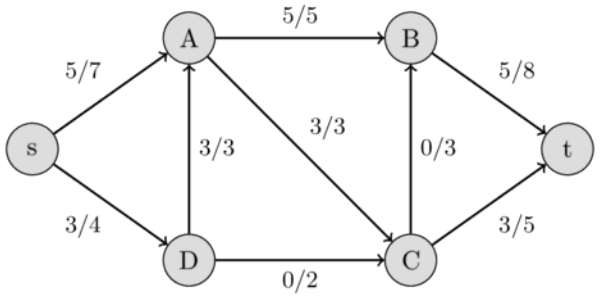
We can find the path $s - A - B - t$ with the residual capacities 7, 5, and 8. Their minimum is 5, therefore we can increase the flow along this path by 5. This gives a flow of 5 for the network.



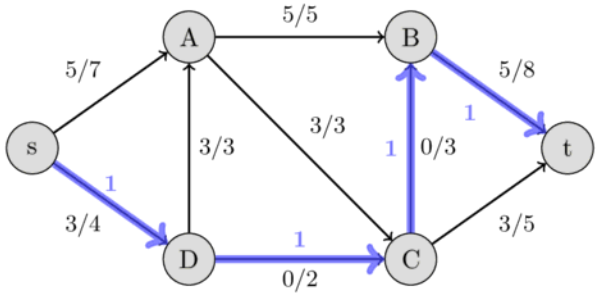


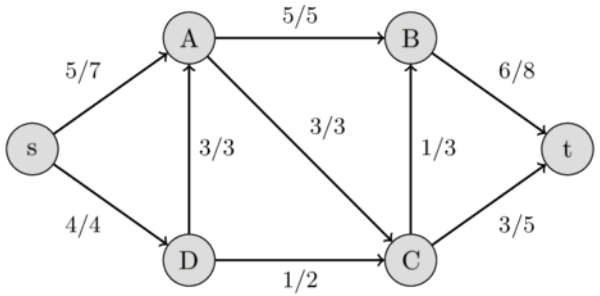
Again we look for an augmenting path, this time we find $s - D - A - C - t$ with the residual capacities 4, 3, 3, and 5. Therefore we can increase the flow by 3 and we get a flow of 8 for the network.



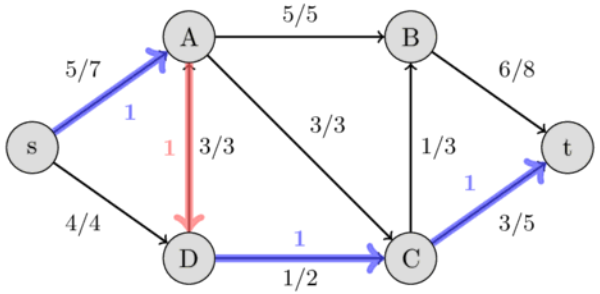


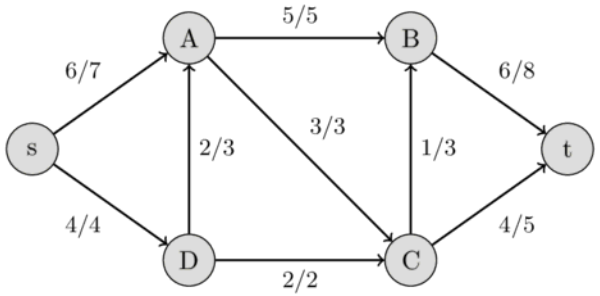
This time we find the path $s - D - C - B - t$ with the residual capacities 1, 2, 3, and 3, and hence, we increase the flow by 1.





This time we find the augmenting path $s - A - D - C - t$ with the residual capacities 2, 3, 1, and 2. We can increase the flow by 1. But this path is very interesting. It includes the reversed edge (A, D). In the original flow network, we are not allowed to send any flow from A to D. But because we already have a flow of 3 from D to A, this is possible. The intuition of it is the following: Instead of sending a flow of 3 from D to A, we only send 2 and compensate this by sending an additional flow of 1 from s to A, which allows us to send an additional flow of 1 along the path $D - C - t$.





Now, it is impossible to find an augmenting path between s and t, therefore this flow of 10 is the maximal possible. We have found the maximal flow.

It should be noted, that the Ford-Fulkerson method doesn't specify a method of finding the augmenting path. Possible approaches are using DFS or BFS which both work in $O(E)$. If all the capacities of the network are integers, then for each augmenting path the flow of the network increases by at least 1 (for more details see Integral flow theorem). Therefore, the complexity of Ford-Fulkerson is $O(EF)$, where F is the maximal flow of the network. In the case of rational capacities, the algorithm will also terminate, but the complexity is not bounded. In the case of irrational capacities, the algorithm might never terminate, and might not even converge to the maximal flow.

Edmonds-Karp algorithm

Edmonds-Karp algorithm is just an implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths. The algorithm was first published by Yefim Dinitz in 1970, and later independently published by Jack Edmonds and Richard Karp in 1972.

The complexity can be given independently of the maximal flow. The algorithm runs in $O(VE^2)$ time, even for irrational capacities. The intuition is, that every time we find an augmenting path one of the edges becomes saturated, and the distance from the edge to s will be longer if it appears later again in an augmenting path. The length of the simple paths is bounded by V.

Implementation

The matrix `capacity` stores the capacity for every pair of vertices. `adj` is the adjacency list of the **undirected graph**, since we have also to use the reversed of directed edges when we are looking for augmenting paths.

The function `maxflow` will return the value of the maximal flow. During the algorithm, the matrix `capacity` will actually store the residual capacity of the network. The value of the flow in each edge will actually not be stored, but it is easy to extend the implementation - by using an additional matrix - to also store the flow and return it.

```
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int>& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == t)
                    return new_flow;
                q.push({next, new_flow});
            }
        }
    }
    return 0;
}

int maxflow(int s, int t) {
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}
```

Integral flow theorem

The theorem simply says, that if every capacity in the network is an integer, then the flow in each edge will be an integer in the maximal flow.

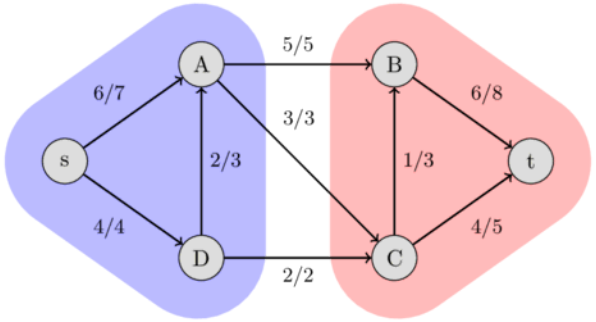
Max-flow min-cut theorem

A **s-t-cut** is a partition of the vertices of a flow network into two sets, such that a set includes the source s and the other one includes the sink t. The capacity of a s-t-cut is defined as the sum of capacities of the edges from the source side to the sink side.

Obviously, we cannot send more flow from s to t than the capacity of any s-t-cut. Therefore, the maximum flow is bounded by the minimum cut capacity.

The max-flow min-cut theorem goes even further. It says that the capacity of the maximum flow has to be equal to the capacity of the minimum cut.

In the following image, you can see the minimum cut of the flow network we used earlier. It shows that the capacity of the cut $\{s, A, D\}$ and $\{B, C, t\}$ is $5 + 3 + 2 = 10$, which is equal to the maximum flow that we found. Other cuts will have a bigger capacity, like the capacity between $\{s, A\}$ and $\{B, C, D, t\}$ is $4 + 3 + 5 = 12$.



A minimum cut can be found after performing a maximum flow computation using the Ford-Fulkerson method. One possible minimum cut is the following: the set of all the vertices that can be reached from s in the residual graph (using edges with positive residual capacity), and the set of all the other vertices. This partition can be easily found using DFS starting at s.

Practice Problems

- Codeforces - Array and Operations
- Codeforces - Red-Blue Graph
- CSES - Download Speed
- CSES - Police Chase
- CSES - School Dance
- CSES - Distinct Routes

Contributors:

jakobkogler (75.0%) DamianArado (13.43%) adamant-pwn (3.24%) infalmo (1.85%) LUTLJS (1.39%) Aryamn (1.39%) Kakalinn (0.93%) obiwac (0.46%) shivensinha4 (0.46%) lm10-piyush (0.46%) ShayekhBinIslam (0.46%) wikku (0.46%) roll-no-1 (0.46%)

Last update: June 8, 2022 Translated From: e-maxx.ru

Floyd-Warshall Algorithm

Given a directed or an undirected weighted graph G with n vertices. The task is to find the length of the shortest path d_{ij} between each pair of vertices i and j.

The graph may have negative weight edges, but no negative weight cycles.

If there is such a negative cycle, you can just traverse this cycle over and over, in each iteration making the cost of the path smaller. So you can make certain paths arbitrarily small, or in other words that shortest path is undefined. That automatically means that an undirected graph cannot have any negative weight edges, as such an edge forms already a negative cycle as you can move back and forth along that edge as long as you like.

This algorithm can also be used to detect the presence of negative cycles. The graph has a negative cycle if at the end of the algorithm, the distance from a vertex v to itself is negative.

This algorithm has been simultaneously published in articles by Robert Floyd and Stephen Warshall in 1962. However, in 1959, Bernard Roy published essentially the same algorithm, but its publication went unnoticed.

Description of the algorithm

The key idea of the algorithm is to partition the process of finding the shortest path between any two vertices to several incremental phases.

Let us number the vertices starting from 1 to n. The matrix of distances is $d[][]$.

Before k-th phase ($k = 1 \dots n$), $d[i][j]$ for any vertices i and j stores the length of the shortest path between the vertex i and vertex j, which contains only the vertices $\{1, 2, \dots, k-1\}$ as internal vertices in the path.

In other words, before k-th phase the value of $d[i][j]$ is equal to the length of the shortest path from vertex i to the vertex j, if this path is allowed to enter only the vertex with numbers smaller than k (the beginning and end of the path are not restricted by this property).

It is easy to make sure that this property holds for the first phase. For $k = 0$, we can fill matrix with $d[i][j] = w_{ij}$ if there exists an edge between i and j with weight w_{ij} and $d[i][j] = \infty$ if there doesn't exist an edge. In practice ∞ will be some high value. As we shall see later, this is a requirement for the algorithm.

Suppose now that we are in the k-th phase, and we want to compute the matrix $d[][]$ so that it meets the requirements for the $(k+1)$-th phase. We have to fix the distances for some vertices pairs (i, j). There are two fundamentally different cases:

- The shortest way from the vertex i to the vertex j with internal vertices from the set $\{1, 2, \dots, k\}$ coincides with the shortest path with internal vertices from the set $\{1, 2, \dots, k-1\}$.

  In this case, $d[i][j]$ will not change during the transition.
- The shortest path with internal vertices from $\{1, 2, \dots, k\}$ is shorter.

  This means that the new, shorter path passes through the vertex k. This means that we can split the shortest path between i and j into two paths: the path between i and k, and the path between k and j. It is clear that both this paths only use internal vertices of $\{1, 2, \dots, k-1\}$ and are the shortest such paths in that respect. Therefore we already have computed the lengths of those paths before, and we can compute the length of the shortest path between i and j as $d[i][k] + d[k][j]$.

Combining these two cases we find that we can recalculate the length of all pairs (i, j) in the k-th phase in the following way:

$$d_{\text{new}}[i][j] = min(d[i][j], d[i][k] + d[k][j])$$

Thus, all the work that is required in the k-th phase is to iterate over all pairs of vertices and recalculate the length of the shortest path between them. As a result, after the n-th phase, the value $d[i][j]$ in the distance matrix is the length of the shortest path between i and j, or is ∞ if the path between the vertices i and j does not exist.

A last remark - we don't need to create a separate distance matrix $d_{\text{new}}[][]$ for temporarily storing the shortest paths of the k-th phase, i.e. all changes can be made directly in the matrix $d[][]$ at any phase. In fact at any k-th phase we are at most improving the distance of any path in the distance matrix, hence we cannot worsen the length of the shortest path for any pair of the vertices that are to be processed in the $(k+1)$-th phase or later.

The time complexity of this algorithm is obviously $O(n^3)$.

Implementation

Let $d[][]$ is a 2D array of size $n \times n$, which is filled according to the 0-th phase as explained earlier. Also we will set $d[i][i] = 0$ for any i at the 0-th phase.

Then the algorithm is implemented as follows:

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
```

```
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

It is assumed that if there is no edge between any two vertices i and j, then the matrix at $d[i][j]$ contains a large number (large enough so that it is greater than the length of any path in this graph). Then this edge will always be unprofitable to take, and the algorithm will work correctly.

However if there are negative weight edges in the graph, special measures have to be taken. Otherwise the resulting values in matrix may be of the form $\infty - 1$, $\infty - 2$, etc., which, of course, still indicates that between the respective vertices doesn't exist a path. Therefore, if the graph has negative weight edges, it is better to write the Floyd-Warshall algorithm in the following way, so that it does not perform transitions using paths that don't exist.

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

Retrieving the sequence of vertices in the shortest path

It is easy to maintain additional information with which it will be possible to retrieve the shortest path between any two given vertices in the form of a sequence of vertices.

For this, in addition to the distance matrix $d[][]$, a matrix of ancestors $p[][]$ must be maintained, which will contain the number of the phase where the shortest distance between two vertices was last modified. It is clear that the number of the phase is nothing more than a vertex in the middle of the desired shortest path. Now we just need to find the shortest path between vertices i and $p[i][j]$, and between $p[i][j]$ and j. This leads to a simple recursive reconstruction algorithm of the shortest path.

The case of real weights

If the weights of the edges are not integer but real, it is necessary to take the errors, which occur when working with float types, into account.

The Floyd-Warshall algorithm has the unpleasant effect, that the errors accumulate very quickly. In fact if there is an error in the first phase of δ, this error may propagate to the second iteration as 2δ, to the third iteration as 4δ, and so on.

To avoid this the algorithm can be modified to take the error (EPS = δ) into account by using following comparison:

```
if (d[i][k] + d[k][j] < d[i][j] - EPS)
    d[i][j] = d[i][k] + d[k][j];
```

The case of negative cycles

Formally the Floyd-Warshall algorithm does not apply to graphs containing negative weight cycle(s). But for all pairs of vertices i and j for which there doesn't exist a path starting at i, visiting a negative cycle, and end at j, the algorithm will still work correctly.

For the pair of vertices for which the answer does not exist (due to the presence of a negative cycle in the path between them), the Floyd algorithm will store any number (perhaps highly negative, but not necessarily) in the distance matrix. However it is possible to improve the Floyd-Warshall algorithm, so that it carefully treats such pairs of vertices, and outputs them, for example as $-\mathrm{INF}$.

This can be done in the following way: let us run the usual Floyd-Warshall algorithm for a given graph. Then a shortest path between vertices i and j does not exist, if and only if, there is a vertex t that is reachable from i and also from j, for which $d[t][t] < 0$.

In addition, when using the Floyd-Warshall algorithm for graphs with negative cycles, we should keep in mind that situations may arise in which distances can get exponentially fast into the negative. Therefore integer overflow must be handled by limiting the minimal distance by some value (e.g. $-\mathrm{INF}$).

To learn more about finding negative cycles in a graph, see the separate article Finding a negative cycle in the graph.

Practice Problems

- UVA: Page Hopping
- SPOJ: Possible Friends
- CODEFORCES: Greg and Graph
- SPOJ: CHICAGO - 106 miles to Chicago
- UVA 10724 - Road Construction
- UVA 117 - The Postal Worker Rings Once
- Codeforces - Traveling Graph
- UVA - 1198 - The Geodetic Set Problem
- UVA - 10048 - Audiophobia

Last update: September 24, 2023 Translated From: e-maxx.ru

Dijkstra Algorithm

You are given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s. This article discusses finding the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves.

This problem is also called **single-source shortest paths problem**.

Algorithm

Here is an algorithm described by the Dutch computer scientist Edsger W. Dijkstra in 1959.

Let's create an array $d[]$ where for each vertex v we store the current length of the shortest path from s to v in $d[v]$. Initially $d[s] = 0$, and for all other vertices this length equals infinity. In the implementation a sufficiently large number (which is guaranteed to be greater than any possible path length) is chosen as infinity.

$$d[v] = \infty, \ v \neq s$$

In addition, we maintain a Boolean array $u[]$ which stores for each vertex v whether it's marked. Initially all vertices are unmarked:

$$u[v] = \text{false}$$

The Dijkstra's algorithm runs for n iterations. At each iteration a vertex v is chosen as unmarked vertex which has the least value $d[v]$:

Evidently, in the first iteration the starting vertex s will be selected.

The selected vertex v is marked. Next, from vertex v **relaxations** are performed: all edges of the form (v, to) are considered, and for each vertex to the algorithm tries to improve the value $d[\text{to}]$. If the length of the current edge equals len, the code for relaxation is:

$$d[\text{to}] = \min(d[\text{to}], d[v] + len)$$

After all such edges are considered, the current iteration ends. Finally, after n iterations, all vertices will be marked, and the algorithm terminates. We claim that the found values $d[v]$ are the lengths of shortest paths from s to all vertices v.

Note that if some vertices are unreachable from the starting vertex s, the values $d[v]$ for them will remain infinite. Obviously, the last few iterations of the algorithm will choose those vertices, but no useful work will be done for them. Therefore, the algorithm can be stopped as soon as the selected vertex has infinite distance to it.

Restoring Shortest Paths

Usually one needs to know not only the lengths of shortest paths but also the shortest paths themselves. Let's see how to maintain sufficient information to restore the shortest path from s to any vertex. We'll maintain an array of predecessors $p[]$ in which for each vertex $v \neq s$, $p[v]$ is the penultimate vertex in the shortest path from s to v. Here we use the fact that if we take the shortest path to some vertex v and remove v from this path, we'll get a path ending in at vertex $p[v]$, and this path will be the shortest for the vertex $p[v]$. This array of predecessors can be used to restore the shortest path to any vertex: starting with v, repeatedly take the predecessor of the current vertex until we reach the starting vertex s to get the required shortest path with vertices listed in reverse order. So, the shortest path P to the vertex v is equal to:

$$P = (s, \dots, p[p[p[v]]], p[p[v]], p[v], v)$$

Building this array of predecessors is very simple: for each successful relaxation, i.e. when for some selected vertex v, there is an improvement in the distance to some vertex to, we update the predecessor vertex for to with vertex v:

$$p[\text{to}] = v$$

Proof

The main assertion on which Dijkstra's algorithm correctness is based is the following:

After any vertex v becomes marked, the current distance to it $d[v]$ is the shortest, and will no longer change.

The proof is done by induction. For the first iteration this statement is obvious: the only marked vertex is s, and the distance to is $d[s] = 0$ is indeed the length of the shortest path to s. Now suppose this statement is true for all previous iterations, i.e. for all already marked vertices; let's prove that it is not violated after the current iteration completes. Let v be the vertex selected in the current iteration, i.e. v

is the vertex that the algorithm will mark. Now we have to prove that $d[v]$ is indeed equal to the length of the shortest path to it $l[v]$.

Consider the shortest path P to the vertex v. This path can be split into two parts: P_1 which consists of only marked nodes (at least the starting vertex s is part of P_1), and the rest of the path P_2 (it may include a marked vertex, but it always starts with an unmarked vertex). Let's denote the first vertex of the path P_2 as p, and the last vertex of the path P_1 as q.

First we prove our statement for the vertex p, i.e. let's prove that $d[p] = l[p]$. This is almost obvious: on one of the previous iterations we chose the vertex q and performed relaxation from it. Since (by virtue of the choice of vertex p) the shortest path to p is the shortest path to q plus edge (p, q), the relaxation from q set the value of $d[p]$ to the length of the shortest path $l[p]$.

Since the edges' weights are non-negative, the length of the shortest path $l[p]$ (which we just proved to be equal to $d[p]$) does not exceed the length $l[v]$ of the shortest path to the vertex v. Given that $l[v] \le d[v]$ (because Dijkstra's algorithm could not have found a shorter way than the shortest possible one), we get the inequality:

$$d[p] = l[p] \le l[v] \le d[v]$$

On the other hand, since both vertices p and v are unmarked, and the current iteration chose vertex v, not p, we get another inequality:

$$d[p] \ge d[v]$$

From these two inequalities we conclude that $d[p] = d[v]$, and then from previously found equations we get:

$$d[v] = l[v]$$

Q.E.D.

Implementation

Dijkstra's algorithm performs n iterations. On each iteration it selects an unmarked vertex v with the lowest value $d[v]$, marks it and checks all the edges (v, to) attempting to improve the value $d[\text{to}]$.

The running time of the algorithm consists of:

- n searches for a vertex with the smallest value $d[v]$ among $O(n)$ unmarked vertices
- m relaxation attempts

For the simplest implementation of these operations on each iteration vertex search requires $O(n)$ operations, and each relaxation can be performed in $O(1)$. Hence, the resulting asymptotic behavior of the algorithm is:

$$O(n^2 + m)$$

This complexity is optimal for dense graph, i.e. when $m \approx n^2$. However in sparse graphs, when m is much smaller than the maximal number of edges n^2, the problem can be solved in $O(n \log n + m)$ complexity. The algorithm and implementation can be found on the article Dijkstra on sparse graphs.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

Here the graph adj is stored as adjacency list: for each vertex v $\text{adj}[v]$ contains the list of edges going from this vertex, i.e. the list of `pair<int,int>` where the first element in the pair is the vertex at the other end of the edge, and the second element is the edge weight.

The function takes the starting vertex s and two vectors that will be used as return values.

First of all, the code initializes arrays: distances $d[]$, labels $u[]$ and predecessors $p[]$. Then it performs n iterations. At each iteration the vertex v is selected which has the smallest distance $d[v]$ among all the unmarked vertices. If the distance to selected vertex v is equal to infinity, the algorithm stops. Otherwise the vertex is marked, and all the edges going out from this vertex are checked. If relaxation along the edge is possible (i.e. distance $d[\text{to}]$ can be improved), the distance $d[\text{to}]$ and predecessor $p[\text{to}]$ are updated.

After performing all the iterations array $d[]$ stores the lengths of the shortest paths to all vertices, and array $p[]$ stores the predecessors of all vertices (except starting vertex s). The path to any vertex t can be restored in the following way:

```
vector<int> restore_path(int s, int t, vector<int> const& p) {
    vector<int> path;

    for (int v = t; v != s; v = p[v])
        path.push_back(v);
    path.push_back(s);

    reverse(path.begin(), path.end());
    return path;
}
```

References

- Edsger Dijkstra. A note on two problems in connexion with graphs [1959]
- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. Introduction to Algorithms [2005]

Practice Problems

- Timus - Ivan's Car [Difficulty:Medium]
- Timus - Sightseeing Trip
- SPOJ - SHPATH [Difficulty:Easy]
- Codeforces - Dijkstra? [Difficulty:Easy]
- Codeforces - Shortest Path
- Codeforces - Jzzhu and Cities
- Codeforces - The Classic Problem
- Codeforces - President and Roads
- Codeforces - Complete The Graph
- TopCoder - SkiResorts

- TopCoder - MaliciousPath
- SPOJ - Ada and Trip
- LA - 3850 - Here We Go(relians) Again
- GYM - Destination Unknown (D)
- UVA 12950 - Even Obsession
- GYM - Journey to Grece (A)
- UVA 13030 - Brain Fry
- UVA 1027 - Toll
- UVA 11377 - Airport Setup
- Codeforces - Dynamic Shortest Path
- UVA 11813 - Shopping
- UVA 11833 - Route Change
- SPOJ - Easy Dijkstra Problem
- LA - 2819 - Cave Raider
- UVA 12144 - Almost Shortest Path
- UVA 12047 - Highest Paid Toll
- UVA 11514 - Batman
- Codeforces - Team Rocket Rises Again
- UVA - 11338 - Minefield
- UVA 11374 - Airport Express
- UVA 11097 - Poor My Problem
- UVA 13172 - The music teacher
- Codeforces - Dirty Arkady's Kitchen
- SPOJ - Delivery Route
- SPOJ - Costly Chess
- CSES - Shortest Routes 1
- CSES - Flight Discount
- CSES - Flight Routes

Contributors:

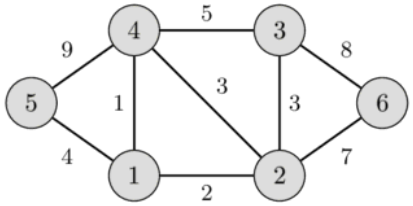
bimalkant-lauhny (31.44%) jakobkogler (23.71%) tcNickolas (16.49%) Morass (11.34%) likecs (8.76%) adamant-pwn (3.09%) Aryamn (1.55%) ChamanAgrawal (1.55%) RodionGork (1.55%) PirateOfAndaman (0.52%)

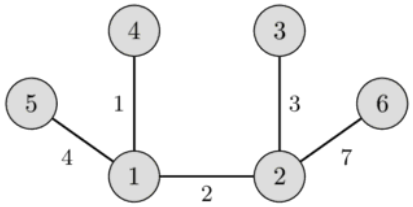
Last update: December 20, 2022 Translated From: e-maxx.ru

Minimum spanning tree - Prim's algorithm

Given a weighted, undirected graph G with n vertices and m edges. You want to find a spanning tree of this graph which connects all vertices and has the least weight (i.e. the sum of weights of edges is minimal). A spanning tree is a set of edges such that any vertex can reach any other by exactly one simple path. The spanning tree with the least weight is called a minimum spanning tree.

In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.





It is easy to see that any spanning tree will necessarily contain $n-1$ edges.

This problem appears quite naturally in a lot of problems. For instance in the following problem: there are n cities and for each pair of cities we are given the cost to build a road between them (or we know that is physically impossible to build a road between them). We have to build roads, such that we can get from each city to every other city, and the cost for building all roads is minimal.

Prim's Algorithm

This algorithm was originally discovered by the Czech mathematician Vojtěch Jarník in 1930. However this algorithm is mostly known as Prim's algorithm after the American mathematician Robert Clay Prim, who rediscovered and republished it in 1957. Additionally Edsger Dijkstra published this algorithm in 1959.

Algorithm description

Here we describe the algorithm in its simplest form. The minimum spanning tree is built gradually by adding edges one at a time. At first the spanning tree consists only of a single vertex (chosen arbitrarily). Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices. Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have $n-1$ edges).

In the end the constructed spanning tree will be minimal. If the graph was originally not connected, then there doesn't exist a spanning tree, so the number of selected edges will be less than $n-1$.

Proof

Let the graph G be connected, i.e. the answer exists. We denote by T the resulting graph found by Prim's algorithm, and by S the minimum spanning tree. Obviously T is indeed a spanning tree and a subgraph of G. We only need to show that the weights of S and T coincide.

Consider the first time in the algorithm when we add an edge to T that is not part of S. Let us denote this edge with e, its ends by a and b, and the set of already selected vertices as V ($a \in V$ and $b \notin V$, or vice versa).

In the minimal spanning tree S the vertices a and b are connected by some path P. On this path we can find an edge f such that one end of f lies in V and the other end doesn't. Since the algorithm chose e instead of f, it means that the weight of f is greater or equal to the weight of e .

We add the edge e to the minimum spanning tree S and remove the edge f. By adding e we created a cycle, and since f was also part of the only cycle, by removing it the resulting graph is again free of cycles. And because we only removed an edge from a cycle, the resulting graph is still connected.

The resulting spanning tree cannot have a larger total weight, since the weight of e was not larger than the weight of f, and it also cannot have a smaller weight since S was a minimum spanning tree. This means that by replacing the edge f with e we generated a different minimum spanning tree. And e has to have the same weight as f.

Thus all the edges we pick in Prim's algorithm have the same weights as the edges of any minimum spanning tree, which means that Prim's algorithm really generates a minimum spanning tree.

Implementation

The complexity of the algorithm depends on how we search for the next minimal edge among the appropriate edges. There are multiple approaches leading to different complexities and different implementations.

Trivial implementations: $O(nm)$ and $O(n^2 + m \log n)$

If we search the edge by iterating over all possible edges, then it takes $O(m)$ time to find the edge with the minimal weight. The total complexity will be $O(nm)$. In the worst case this is $O(n^3)$, really slow.

This algorithm can be improved if we only look at one edge from each already selected vertex. For example we can sort the edges from each vertex in ascending order of their weights, and store a pointer to the first valid edge (i.e. an edge that goes to an non-selected vertex). Then after finding and selecting the minimal edge, we update the pointers. This give a complexity of $O(n^2 + m)$, and for sorting the edges an additional $O(m \log n)$, which gives the complexity $O(n^2 \log n)$ in the worst case.

Below we consider two slightly different algorithms, one for dense and one for sparse graphs, both with a better complexity.

Dense graphs: $O(n^2)$

We approach this problem from a different angle: for every not yet selected vertex we will store the minimum edge to an already selected vertex.

Then during a step we only have to look at these minimum weight edges, which will have a complexity of $O(n)$.

After adding an edge some minimum edge pointers have to be recalculated. Note that the weights only can decrease, i.e. the minimal weight edge of every not yet selected vertex might stay the same, or it will be updated by an edge to the newly selected vertex. Therefore this phase can also be done in $O(n)$.

Thus we received a version of Prim's algorithm with the complexity $O(n^2)$.

In particular this implementation is very convenient for the Euclidean Minimum Spanning Tree problem: we have n points on a plane and the distance between each pair of points is the Euclidean distance between them, and we want to find a minimum spanning tree for this complete graph. This task can be solved by the described algorithm in $O(n^2)$ time and $O(n)$ memory, which is not possible with Kruskal's algorithm.

```
int n;
vector<vector<int>> adj; // adjacency matrix of graph
const int INF = 1000000000; // weight INF means there is no edge

struct Edge {
    int w = INF, to = -1;
};

void prim() {
    int total_weight = 0;
    vector<bool> selected(n, false);
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
```

```
                min_e[to] = {adj[v][to], v};
            }
        }

        cout << total_weight << endl;
    }
```

The adjacency matrix `adj[][]` of size $n \times n$ stores the weights of the edges, and it uses the weight `INF` if there doesn't exist an edge between two vertices. The algorithm uses two arrays: the flag `selected[]`, which indicates which vertices we already have selected, and the array `min_e[]` which stores the edge with minimal weight to an selected vertex for each not-yet-selected vertex (it stores the weight and the end vertex). The algorithm does n steps, in each iteration the vertex with the smallest edge weight is selected, and the `min_e[]` of all other vertices gets updated.

Sparse graphs: $O(m \log n)$

In the above described algorithm it is possible to interpret the operations of finding the minimum and modifying some values as set operations. These two classical operations are supported by many data structure, for example by `set` in C++ (which are implemented via red-black trees).

The main algorithm remains the same, but now we can find the minimum edge in $O(\log n)$ time. On the other hand recomputing the pointers will now take $O(n \log n)$ time, which is worse than in the previous algorithm.

But when we consider that we only need to update $O(m)$ times in total, and perform $O(n)$ searches for the minimal edge, then the total complexity will be $O(m \log n)$. For sparse graphs this is better than the above algorithm, but for dense graphs this will be slower.

```
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to);
    }
};

int n;
vector<vector<Edge>> adj;

void prim() {
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
```



```
    vector<bool> selected(n, false);
    for (int i = 0; i < n; ++i) {
        if (q.empty()) {
            cout << "No MST!" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        selected[v] = true;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v]) {
            if (!selected[e.to] && e.w < min_e[e.to].w) {
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }

    cout << total_weight << endl;
}
```

Here the graph is represented via a adjacency list `adj[]`, where `adj[v]` contains all edges (in form of weight and target pairs) for the vertex `v`. `min_e[v]` will store the weight of the smallest edge from vertex `v` to an already selected vertex (again in the form of a weight and target pair). In addition the queue `q` is filled with all not yet selected vertices in the order of increasing weights `min_e`. The algorithm does `n` steps, on each of which it selects the vertex `v` with the smallest weight `min_e` (by extracting it from the beginning of the queue), and then looks through all the edges from this vertex and updates the values in `min_e` (during an update we also need to also remove the old edge from the queue `q` and put in the new edge).

Contributors:
jakobkogler (94.31%) adamant-pwn (2.84%) mrpandey (0.95%) ekanshi258 (0.47%)
jyterencekim (0.47%) Naman-Bhalla (0.47%) wikku (0.47%)

Last update: September 23, 2023 Translated From: e-maxx.ru

2-SAT

SAT (Boolean satisfiability problem) is the problem of assigning Boolean values to variables to satisfy a given Boolean formula. The Boolean formula will usually be given in CNF (conjunctive normal form), which is a conjunction of multiple clauses, where each clause is a disjunction of literals (variables or negation of variables). 2-SAT (2-satisfiability) is a restriction of the SAT problem, in 2-SAT every clause has exactly two literals. Here is an example of such a 2-SAT problem. Find an assignment of a, b, c such that the following formula is true:

$$(a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$

SAT is NP-complete, there is no known efficient solution for it. However 2SAT can be solved efficiently in $O(n + m)$ where n is the number of variables and m is the number of clauses.

Algorithm:

First we need to convert the problem to a different form, the so-called implicative normal form. Note that the expression $a \vee b$ is equivalent to $\neg a \Rightarrow b \wedge \neg b \Rightarrow a$ (if one of the two variables is false, then the other one must be true).

We now construct a directed graph of these implications: for each variable x there will be two vertices v_x and $v_{\neg x}$. The edges will correspond to the implications.

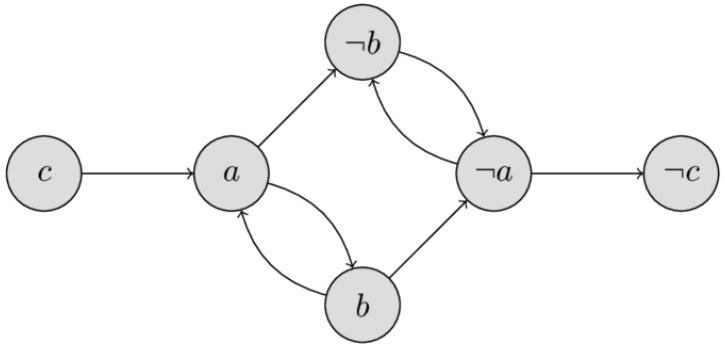
Let's look at the example in 2-CNF form:

$$(a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$

The oriented graph will contain the following vertices and edges:

$$\begin{array}{cccc} \neg a \Rightarrow \neg b & a \Rightarrow b & a \Rightarrow \neg b & \neg a \Rightarrow \neg c \\ b \Rightarrow a & \neg b \Rightarrow \neg a & b \Rightarrow \neg a & c \Rightarrow a \end{array}$$

You can see the implication graph in the following image:



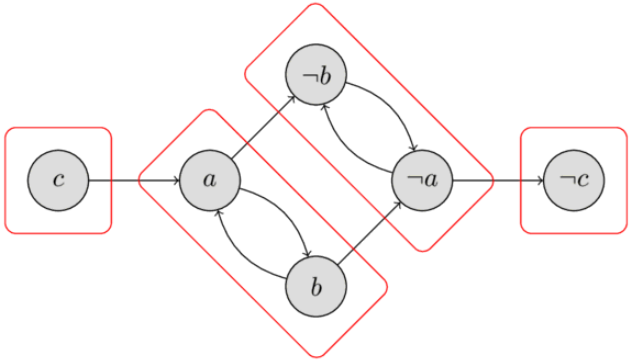
It is worth paying attention to the property of the implication graph: if there is an edge $a \Rightarrow b$, then there also is an edge $\neg b \Rightarrow \neg a$.

Also note, that if x is reachable from $\neg x$, and $\neg x$ is reachable from x, then the problem has no solution. Whatever value we choose for the variable x, it will always end in a contradiction - if x will be assigned true then the implication tell us that $\neg x$ should also be true and visa versa. It turns out, that this condition is not only necessary, but also sufficient. We will prove this in a few paragraphs below. First recall, if a vertex is reachable from a second one, and the second one is reachable from the first one, then these two vertices are in the same strongly connected component. Therefore we can formulate the criterion for the existence of a solution as follows:

In order for this 2-SAT problem to have a solution, it is necessary and sufficient that for any variable x the vertices x and $\neg x$ are in different strongly connected components of the strong connection of the implication graph.

This criterion can be verified in $O(n + m)$ time by finding all strongly connected components.

The following image shows all strongly connected components for the example. As we can check easily, neither of the four components contain a vertex x and its negation $\neg x$, therefore the example has a solution. We will learn in the next paragraphs how to compute a valid assignment, but just for demonstration purposes the solution $a = \text{false}$, $b = \text{false}$, $c = \text{false}$ is given.



Now we construct the algorithm for finding the solution of the 2-SAT problem on the assumption that the solution exists.

Note that, in spite of the fact that the solution exists, it can happen that $\neg x$ is reachable from x in the implication graph, or that (but not simultaneously) x is reachable from $\neg x$. In that case the choice of either true or false for x will lead to a contradiction, while the choice of the other one will not. Let's learn how to choose a value, such that we don't generate a contradiction.

Let us sort the strongly connected components in topological order (i.e. $\text{comp}[v] \leq \text{comp}[u]$ if there is a path from v to u) and let $\text{comp}[v]$ denote the index of strongly connected component to which the vertex v belongs. Then, if $\text{comp}[x] < \text{comp}[\neg x]$ we assign x with false and true otherwise.

Let us prove that with this assignment of the variables we do not arrive at a contradiction. Suppose x is assigned with true. The other case can be proven in a similar way.

First we prove that the vertex x cannot reach the vertex $\neg x$. Because we assigned true it has to hold that the index of strongly connected component of x is greater than the index of the component of $\neg x$. This means that $\neg x$ is located on the left of the component containing x, and the later vertex cannot reach the first.

Secondly we prove that there doesn't exist a variable y, such that the vertices y and $\neg y$ are both reachable from x in the implication graph. This would cause a contradiction, because $x = \text{true}$ implies that $y = \text{true}$ and $\neg y = \text{true}$. Let us prove this by contradiction. Suppose that y and $\neg y$ are both reachable from x, then by the property of the implication graph $\neg x$ is reachable from both y and $\neg y$. By transitivity this results that $\neg x$ is reachable by x, which contradicts the assumption.

So we have constructed an algorithm that finds the required values of variables under the assumption that for any variable x the vertices x and $\neg x$ are in different strongly connected components. Above showed the correctness of this algorithm. Consequently we simultaneously proved the above criterion for the existence of a solution.

Implementation:

Now we can implement the entire algorithm. First we construct the graph of implications and find all strongly connected components. This can be accomplished with Kosaraju's algorithm in $O(n + m)$ time. In the second traversal of the graph Kosaraju's algorithm visits the strongly connected components in topological order, therefore it is easy to compute $\text{comp}[v]$ for each vertex v.

Afterwards we can choose the assignment of x by comparing $\text{comp}[x]$ and $\text{comp}[\neg x]$. If $\text{comp}[x] = \text{comp}[\neg x]$ we return false to indicate that there doesn't exist a valid assignment that satisfies the 2-SAT problem.

Below is the implementation of the solution of the 2-SAT problem for the already constructed graph of implication adj and the transpose graph adj^T (in which the direction of each edge is reversed). In the graph the vertices with indices $2k$ and $2k + 1$ are the two vertices corresponding to variable k with $2k + 1$ corresponding to the negated variable.

```
int n;
vector<vector<int>> adj, adj_t;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;

void dfs1(int v) {
    used[v] = true;
    for (int u : adj[v]) {
        if (!used[u])
            dfs1(u);
    }
    order.push_back(v);
}

void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : adj_t[v]) {
        if (comp[u] == -1)
            dfs2(u, cl);
    }
}

bool solve_2SAT() {
    order.clear();
    used.assign(n, false);
    for (int i = 0; i < n; ++i) {
        if (!used[i])
            dfs1(i);
    }

    comp.assign(n, -1);
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }

    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])
            return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true;
}

void add_disjunction(int a, bool na, int b, bool nb) {
```

```
    // na and nb signify whether a and b are to be negated
    a = 2*a ^ na;
    b = 2*b ^ nb;
    int neg_a = a ^ 1;
    int neg_b = b ^ 1;
    adj[neg_a].push_back(b);
    adj[neg_b].push_back(a);
    adj_t[b].push_back(neg_a);
    adj_t[a].push_back(neg_b);
}
```

Practice Problems

- Codeforces: The Door Problem
- Kattis: Illumination
- UVA: Rectangles
- Codeforces : Radio Stations
- CSES : Giant Pizza

Contributors:
jakobkogler (88.82%) adamant-pwn (3.53%) jatin-code777 (2.35%) matbensch (1.18%) vatsalsharma376 (1.18%)
sumitrawat10 (0.59%) aiifabbf (0.59%) Aryamn (0.59%) roll-no-1 (0.59%) wikku (0.59%)

Last update: June 8, 2022 Translated From: e-maxx.ru

Lowest Common Ancestor - $O(\sqrt{N})$ and $O(\log N)$ with $O(N)$ preprocessing

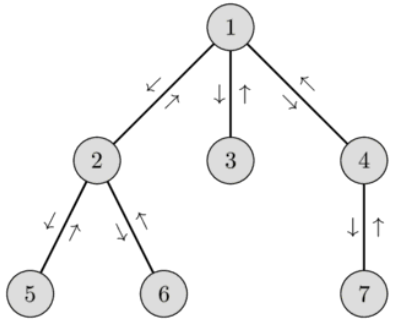
Given a tree G. Given queries of the form (v_1, v_2), for each query you need to find the lowest common ancestor (or least common ancestor), i.e. a vertex v that lies on the path from the root to v_1 and the path from the root to v_2, and the vertex should be the lowest. In other words, the desired vertex v is the most bottom ancestor of v_1 and v_2. It is obvious that their lowest common ancestor lies on a shortest path from v_1 and v_2. Also, if v_1 is the ancestor of v_2, v_1 is their lowest common ancestor.

The Idea of the Algorithm

Before answering the queries, we need to **preprocess** the tree. We make a DFS traversal starting at the root and we build a list euler which stores the order of the vertices that we visit (a vertex is added to the list when we first visit it, and after the return of the DFS traversals to its children). This is also called an Euler tour of the tree. It is clear that the size of this list will be $O(N)$. We also need to build an array $\text{first}[0..N-1]$ which stores for each vertex i its first occurrence in euler. That is, the first position in euler such that $\text{euler}[\text{first}[i]] = i$. Also by using the DFS we can find the height of each node (distance from root to it) and store it in the array $\text{height}[0..N-1]$.

So how can we answer queries using the Euler tour and the additional two arrays? Suppose the query is a pair of v_1 and v_2. Consider the vertices that we visit in the Euler tour between the first visit of v_1 and the first visit of v_2. It is easy to see, that the $\text{LCA}(v_1, v_2)$ is the vertex with the lowest height on this path. We already noticed, that the LCA has to be part of the shortest path between v_1 and v_2. Clearly it also has to be the vertex with the smallest height. And in the Euler tour we essentially use the shortest path, except that we additionally visit all subtrees that we find on the path. But all vertices in these subtrees are lower in the tree than the LCA and therefore have a larger height. So the $\text{LCA}(v_1, v_2)$ can be uniquely determined by finding the vertex with the smallest height in the Euler tour between $\text{first}(v_1)$ and $\text{first}(v_2)$.

Let's illustrate this idea. Consider the following graph and the Euler tour with the corresponding heights:



Vertices:	1	2	5	2	6	2	1	3	1	4	7	4	1
Heights:	1	2	3	2	3	2	1	2	1	2	3	2	1

The tour starting at vertex 6 and ending at 4 we visit the vertices $[6, 2, 1, 3, 1, 4]$. Among those vertices the vertex 1 has the lowest height, therefore $\text{LCA}(6, 4) = 1$.

To recap: to answer a query we just need **to find the vertex with smallest height** in the array euler in the range from $\text{first}[v_1]$ to $\text{first}[v_2]$. Thus, **the LCA problem is reduced to the RMQ problem** (finding the minimum in an range problem).

Using Sqrt-Decomposition, it is possible to obtain a solution answering each query in $O(\sqrt{N})$ with preprocessing in $O(N)$ time.

Using a Segment Tree you can answer each query in $O(\log N)$ with preprocessing in $O(N)$ time.

Since there will almost never be any update to the stored values, a Sparse Table might be a better choice, allowing $O(1)$ query answering with $O(N \log N)$ build time.

Implementation

In the following implementation of the LCA algorithm a Segment Tree is used.

```
struct LCA {
    vector<int> height, euler, first, segtree;
    vector<bool> visited;
    int n;

    LCA(vector<vector<int>> &adj, int root = 0) {
        n = adj.size();
```

```
        height.resize(n);
        first.resize(n);
        euler.reserve(n * 2);
        visited.assign(n, false);
        dfs(adj, root);
        int m = euler.size();
        segtree.resize(m * 4);
        build(1, 0, m - 1);
    }

    void dfs(vector<vector<int>> &adj, int node, int h = 0) {
        visited[node] = true;
        height[node] = h;
        first[node] = euler.size();
        euler.push_back(node);
        for (auto to : adj[node]) {
            if (!visited[to]) {
                dfs(adj, to, h + 1);
                euler.push_back(node);
            }
        }
    }

    void build(int node, int b, int e) {
        if (b == e) {
            segtree[node] = euler[b];
        } else {
            int mid = (b + e) / 2;
            build(node << 1, b, mid);
            build(node << 1 | 1, mid + 1, e);
            int l = segtree[node << 1], r = segtree[node << 1 | 1];
            segtree[node] = (height[l] < height[r]) ? l : r;
        }
    }

    int query(int node, int b, int e, int L, int R) {
        if (b > R || e < L)
            return -1;
        if (b >= L && e <= R)
            return segtree[node];
        int mid = (b + e) >> 1;

        int left = query(node << 1, b, mid, L, R);
        int right = query(node << 1 | 1, mid + 1, e, L, R);
        if (left == -1) return right;
        if (right == -1) return left;
        return height[left] < height[right] ? left : right;
    }

    int lca(int u, int v) {
        int left = first[u], right = first[v];
        if (left > right)
            swap(left, right);
        return query(1, 0, euler.size() - 1, left, right);
    }
};
```

Last update: June 8, 2022 Translated From: e-maxx.ru

Optimal schedule of jobs given their deadlines and durations

Suppose, we have a set of jobs, and we are aware of every job's deadline and its duration. The execution of a job cannot be interrupted prior to its ending. It is required to create such a schedule to accomplish the biggest number of jobs.

Solving

The algorithm of the solving is **greedy**. Let's sort all the jobs by their deadlines and look at them in descending order. Also, let's create a queue q, in which we'll gradually put the jobs and extract one with the least run-time (for instance, we can use set or priority_queue). Initially, q is empty.

Suppose, we're looking at the i-th job. First of all, let's put it into q. Let's consider the period of time between the deadline of i-th job and the deadline of $i-1$-th job. That is the segment of some length T. We will extract jobs from q (in their left duration ascending order) and execute them until the whole segment T is filled. Important: if at any moment of time the extracted job can only be partly executed until segment T is filled, then we execute this job partly just as far as possible, i.e., during the T-time, and we put the remaining part of a job back into q.

On the algorithm's completion we'll choose the optimal solution (or, at least, one of several solutions). The running time of algorithm is $O(n \log n)$.

Implementation

The following function takes a vector of jobs (consisting of a deadline, a duration, and the job's index) and computes a vector containing all indices of the used jobs in the optimal schedule. Notice that you still need to sort these jobs by their deadline, if you want to write down the plan explicitly.

```
struct Job {
    int deadline, duration, idx;

    bool operator<(Job o) const {
        return deadline < o.deadline;
    }
};

vector<int> compute_schedule(vector<Job> jobs) {
    sort(jobs.begin(), jobs.end());

    set<pair<int,int>> s;
    vector<int> schedule;
    for (int i = jobs.size()-1; i >= 0; i--) {
        int t = jobs[i].deadline - (i ? jobs[i-1].deadline : 0);
        s.insert(make_pair(jobs[i].duration, jobs[i].idx));
        while (t && !s.empty()) {
            auto it = s.begin();
            if (it->first <= t) {
                t -= it->first;
                schedule.push_back(it->second);
            } else {
                s.insert(make_pair(it->first - t, it->second));
                t = 0;
            }
            s.erase(it);
        }
    }
    return schedule;
}
```

Contributors:

jakobkogler (60.0%) tcNickolas (29.09%) adamant-pwn (10.91%)

2/23/24, 8:35 PM Optimal schedule of jobs given their deadlines and durations - Algorithms for Competitive Programming

Last update: June 8, 2022 Translated From: e-maxx.ru

Kth order statistic in $O(N)$

Given an array A of size N and a number K. The problem is to find K-th largest number in the array, i.e., K-th order statistic.

The basic idea - to use the idea of quick sort algorithm. Actually, the algorithm is simple, it is more difficult to prove that it runs in an average of $O(N)$, in contrast to the quick sort.

Implementation (not recursive)

```
template <class T>
T order_statistics (std::vector<T> a, unsigned n, unsigned k)
{
    using std::swap;
    for (unsigned l=1, r=n; ; )
    {
        if (r <= l+1)
        {
            // the current part size is either 1 or 2, so it is easy to find the
answer
            if (r == l+1 && a[r] < a[l])
                swap (a[l], a[r]);
            return a[k];
        }

        // ordering a[l], a[l+1], a[r]
        unsigned mid = (l + r) >> 1;
        swap (a[mid], a[l+1]);
        if (a[l] > a[r])
            swap (a[l], a[r]);
        if (a[l+1] > a[r])
            swap (a[l+1], a[r]);
        if (a[l] > a[l+1])
            swap (a[l], a[l+1]);

        // performing division
        // barrier is a[l + 1], i.e. median among a[l], a[l + 1], a[r]
        unsigned
            i = l+1,
            j = r;
        const T
            cur = a[l+1];
        for (;;)
        {
            while (a[++i] < cur) ;
            while (a[--j] > cur) ;
            if (i > j)
                break;
            swap (a[i], a[j]);
        }

        // inserting the barrier
        a[l+1] = a[j];
        a[j] = cur;

        // we continue to work in that part, which must contain the required
element
        if (j >= k)
            r = j-1;
        if (j <= k)
            l = i;
    }
}
```

Notes

- The randomized algorithm above is named quickselect. You should do random shuffle on A before calling it or use a random element as a barrier for it to run properly. There are also deterministic algorithms that solve the specified problem in linear time, such as median of medians.
- A deterministic linear solution is implemented in C++ standard library as std::nth_element.
- Finding K smallest elements can be reduced to finding K-th element with a linear overhead, as they're exactly the elements that are smaller than K-th.

Practice Problems

- CODECHEF: Median

Contributors:

likecs (60.53%) ErzhanDS (15.79%) adamant-pwn (13.16%) jakobkogler (5.26%) dufferzafar (3.95%) shark_s (1.32%)

2/23/24, 8:34 PM K-th order statistic in O(N) - Algorithms for Competitive Programming

Last update: February 10, 2024 Translated From: e-maxx.ru

Longest increasing subsequence

We are given an array with n numbers: $a[0 \dots n-1]$. The task is to find the longest, strictly increasing, subsequence in a.

Formally we look for the longest sequence of indices $i_1, \dots i_k$ such that

$$i_1 < i_2 < \dots < i_k, \quad a[i_1] < a[i_2] < \dots < a[i_k]$$

In this article we discuss multiple algorithms for solving this task. Also we will discuss some other problems, that can be reduced to this problem.

Solution in $O(n^2)$ with dynamic programming

Dynamic programming is a very general technique that allows to solve a huge class of problems. Here we apply the technique for our specific task.

First we will search only for the **length** of the longest increasing subsequence, and only later learn how to restore the subsequence itself.

Finding the length

To accomplish this task, we define an array $d[0 \dots n-1]$, where $d[i]$ is the length of the longest increasing subsequence that ends in the element at index i.

Example

$$\begin{aligned} a &= \{8, 3, 4, 6, 5, 2, 0, 7, 9, 1\} \\ d &= \{1, 1, 2, 3, 3, 1, 1, 4, 5, 2\} \end{aligned}$$

The longest increasing subsequence that ends at index 4 is $\{3, 4, 5\}$ with a length of 3, the longest ending at index 8 is either $\{3, 4, 5, 7, 9\}$ or $\{3, 4, 6, 7, 9\}$, both having length 5, and the longest ending at index 9 is $\{0, 1\}$ having length 2.

We will compute this array gradually: first $d[0]$, then $d[1]$, and so on. After this array is computed, the answer to the problem will be the maximum value in the array $d[]$.

So let the current index be i. I.e. we want to compute the value $d[i]$ and all previous values $d[0], \dots, d[i-1]$ are already known. Then there are two options:

- $d[i] = 1$: the required subsequence consists only of the element $a[i]$.
- $d[i] > 1$: The subsequence will end at $a[i]$, and right before it will be some number $a[j]$ with $j < i$ and $a[j] < a[i]$.

  It's easy to see, that the subsequence ending in $a[j]$ will itself be one of the longest increasing subsequences that ends in $a[j]$. The number $a[i]$ just extends that longest increasing subsequence by one number.

  Therefore, we can just iterate over all $j < i$ with $a[j] < a[i]$, and take the longest sequence that we get by appending $a[i]$ to the longest increasing subsequence ending in $a[j]$. The longest increasing subsequence ending in $a[j]$ has length $d[j]$, extending it by one gives the length $d[j] + 1$.

$$d[i] = \max_{\substack{j<i \\ a[j]<a[i]}} (d[j] + 1)$$

If we combine these two cases we get the final answer for $d[i]$:

$$d[i] = \max \left(1, \max_{\substack{j<i \\ a[j]<a[i]}} (d[j] + 1) \right)$$

Implementation

Here is an implementation of the algorithm described above, which computes the length of the longest increasing subsequence.

```
int lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i])
                d[i] = max(d[i], d[j] + 1);
        }
    }

    int ans = d[0];
    for (int i = 1; i < n; i++) {
        ans = max(ans, d[i]);
    }
    return ans;
}
```

Restoring the subsequence

So far we only learned how to find the length of the subsequence, but not how to find the subsequence itself.

To be able to restore the subsequence we generate an additional auxiliary array $p[0 \dots n-1]$ that we will compute alongside the array $d[]$. $p[i]$ will be the index j of the second last element in the longest increasing subsequence ending in i. In other words the index $p[i]$ is the same index j at which the highest value $d[i]$ was obtained. This auxiliary array $p[]$ points in some sense to the ancestors.

Then to derive the subsequence, we just start at the index i with the maximal $d[i]$, and follow the ancestors until we deduced the entire subsequence, i.e. until we reach the element with $d[i] = 1$.

Implementation of restoring

We will change the code from the previous sections a little bit. We will compute the array $p[]$ alongside $d[]$, and afterwards compute the subsequence.

For convenience we originally assign the ancestors with $p[i] = -1$. For elements with $d[i] = 1$, the ancestors value will remain -1, which will be slightly more convenient for restoring the subsequence.

```
vector<int> lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1), p(n, -1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i] && d[i] < d[j] + 1) {
                d[i] = d[j] + 1;
                p[i] = j;
            }
        }
    }

    int ans = d[0], pos = 0;
    for (int i = 1; i < n; i++) {
        if (d[i] > ans) {
            ans = d[i];
            pos = i;
        }
    }

    vector<int> subseq;
    while (pos != -1) {
        subseq.push_back(a[pos]);
        pos = p[pos];
    }
    reverse(subseq.begin(), subseq.end());
    return subseq;
}
```

Alternative way of restoring the subsequence

It is also possible to restore the subsequence without the auxiliary array $p[]$. We can simply recalculate the current value of $d[i]$ and also see how the maximum was reached.

This method leads to a slightly longer code, but in return we save some memory.

Solution in $O(n \log n)$ with dynamic programming and binary search

In order to obtain a faster solution for the problem, we construct a different dynamic programming solution that runs in $O(n^2)$, and then later improve it to $O(n \log n)$.

We will use the dynamic programming array $d[0 \dots n]$. This time $d[l]$ doesn't corresponds to the element $a[i]$ or to an prefix of the array. $d[l]$ will be the smallest element at which an increasing subsequence of length l ends.

Initially we assume $d[0] = -\infty$ and for all other lengths $d[l] = \infty$.

We will again gradually process the numbers, first $a[0]$, then $a[1]$, etc, and in each step maintain the array $d[]$ so that it is up to date.

Example

Given the array $a = \{8, 3, 4, 6, 5, 2, 0, 7, 9, 1\}$, here are all their prefixes and their dynamic programming array. Notice, that the values of the array don't always change at the end.

$$\begin{aligned}
\text{prefix} &= \{\} & d &= \{-\infty, \infty, \dots\} \\
\text{prefix} &= \{8\} & d &= \{-\infty, 8, \infty, \dots\} \\
\text{prefix} &= \{8, 3\} & d &= \{-\infty, 3, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4\} & d &= \{-\infty, 3, 4, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4, 6\} & d &= \{-\infty, 3, 4, 6, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4, 6, 5\} & d &= \{-\infty, 3, 4, 5, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4, 6, 5, 2\} & d &= \{-\infty, 2, 4, 5, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4, 6, 5, 2, 0\} & d &= \{-\infty, 0, 4, 5, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4, 6, 5, 2, 0, 7\} & d &= \{-\infty, 0, 4, 5, 7, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4, 6, 5, 2, 0, 7, 9\} & d &= \{-\infty, 0, 4, 5, 7, 9, \infty, \dots\} \\
\text{prefix} &= \{8, 3, 4, 6, 5, 2, 0, 7, 9, 1\} & d &= \{-\infty, 0, 1, 5, 7, 9, \infty, \dots\}
\end{aligned}$$

When we process $a[i]$, we can ask ourselves. What have the conditions to be, that we write the current number $a[i]$ into the $d[0 \dots n]$ array?

We set $d[l] = a[i]$, if there is a longest increasing sequence of length l that ends in $a[i]$, and there is no longest increasing sequence of length l that ends in a smaller number. Similar to the previous approach, if we remove the number $a[i]$ from the longest increasing sequence of length l, we get another longest increasing sequence of length $l-1$. So we want to extend a longest increasing sequence of length $l-1$ by the number $a[i]$, and obviously the longest increasing sequence of length $l-1$ that ends with the smallest element will work the best, in other words the sequence of length $l-1$ that ends in element $d[l-1]$.

There is a longest increasing sequence of length $l-1$ that we can extend with the number $a[i]$, exactly if $d[l-1] < a[i]$. So we can just iterate over each length l, and check if we can extend a longest increasing sequence of length $l-1$ by checking the criteria.

Additionally we also need to check, if we maybe have already found a longest increasing sequence of length l with a smaller number at the end. So we only update if $a[i] < d[l]$.

After processing all the elements of $a[]$ the length of the desired subsequence is the largest l with $d[l] < \infty$.

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        for (int l = 1; l <= n; l++) {
            if (d[l-1] < a[i] && a[i] < d[l])
                d[l] = a[i];
        }
    }

    int ans = 0;
    for (int l = 0; l <= n; l++) {
        if (d[l] < INF)
            ans = l;
    }
    return ans;
}
```

We now make two important observations.

1. The array d will always be sorted: $d[l-1] < d[l]$ for all $i = 1 \dots n$.

   This is trivial, as you can just remove the last element from the increasing subsequence of length l, and you get a increasing subsequence of length $l-1$ with a smaller ending number.
2. The element $a[i]$ will only update at most one value $d[l]$.

   This follows immediately from the above implementation. There can only be one place in the array with $d[l-1] < a[i] < d[l]$.

Thus we can find this element in the array $d[]$ using binary search in $O(\log n)$. In fact we can simply look in the array $d[]$ for the first number that is strictly greater than $a[i]$, and we try to update this element in the same way as the above implementation.

Implementation

This gives us the improved $O(n \log n)$ implementation:

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[l-1] < a[i] && a[i] < d[l])
            d[l] = a[i];
    }

    int ans = 0;
    for (int l = 0; l <= n; l++) {
        if (d[l] < INF)
            ans = l;
    }
    return ans;
}
```

Restoring the subsequence

It is also possible to restore the subsequence using this approach. This time we have to maintain two auxiliary arrays. One that tells us the index of the elements in $d[]$. And again we have to create an array of "ancestors" $p[i]$. $p[i]$ will be the index of the previous element for the optimal subsequence ending in element i.

It's easy to maintain these two arrays in the course of iteration over the array $a[]$ alongside the computations of $d[]$. And at the end it is not difficult to restore the desired subsequence using these arrays.

Solution in $O(n \log n)$ with data structures

Instead of the above method for computing the longest increasing subsequence in $O(n \log n)$ we can also solve the problem in a different way: using some simple data structures.

Let's go back to the first method. Remember that $d[i]$ is the value $d[j]+1$ with $j < i$ and $a[j] < a[i]$.

Thus if we define an additional array $t[]$ such that

$$t[a[i]] = d[i],$$

then the problem of computing the value $d[i]$ is equivalent to finding the **maximum value in a prefix** of the array $t[]$:

$$d[i] = \max\left(t[0 \dots a[i]-1] + 1\right)$$

The problem of finding the maximum of a prefix of an array (which changes) is a standard problem that can be solved by many different data structures. For instance we can use a Segment tree or a Fenwick tree.

This method has obviously some **shortcomings**: in terms of length and complexity of the implementation this approach will be worse than the method using binary search. In addition if the input numbers $a[i]$ are especially large, then we would have to use some tricks, like compressing the numbers (i.e. renumber them

from 0 to $n-1$), or use a dynamic segment tree (only generate the branches of the tree that are important). Otherwise the memory consumption will be too high.

On the other hand this method has also some **advantages**: with this method you don't have to think about any tricky properties in the dynamic programming solution. And this approach allows us to generalize the problem very easily (see below).

Related tasks

Here are several problems that are closely related to the problem of finding the longest increasing subsequence.

Longest non-decreasing subsequence

This is in fact nearly the same problem. Only now it is allowed to use identical numbers in the subsequence.

The solution is essentially also nearly the same. We just have to change the inequality signs, and make a slightly modification to the binary search.

Number of longest increasing subsequences

We can use the first discussed method, either the $O(n^2)$ version or the version using data structures. We only have to additionally store in how many ways we can obtain longest increasing subsequences ending in the values $d[i]$.

The number of ways to form a longest increasing subsequences ending in $a[i]$ is the sum of all ways for all longest increasing subsequences ending in j where $d[j]$ is maximal. There can be multiple such j, so we need to sum all of them.

Using a Segment tree this approach can also be implemented in $O(n \log n)$.

It is not possible to use the binary search approach for this task.

Smallest number of non-increasing subsequences covering a sequence

For a given array with n numbers $a[0 \dots n-1]$ we have to colorize the numbers in the smallest number of colors, so that each color forms a non-increasing subsequence.

To solve this, we notice that the minimum number of required colors is equal to the length of the longest increasing subsequence.

Proof: We need to prove the **duality** of these two problems.

Let's denote by x the length of the longest increasing subsequence and by y the least number of non-increasing subsequences that form a cover. We need to prove that $x = y$.

It is clear that $y < x$ is not possible, because if we have x strictly increasing elements, than no two can be part of the same non-increasing subsequence. Therefore we have $y \geq x$.

We now show that $y > x$ is not possible by contradiction. Suppose that $y > x$. Then we consider any optimal set of y non-increasing subsequences. We transform this in set in the following way: as long as there are two such subsequences such that the first begins before the second subsequence, and the first sequence start with a number greater than or equal to the second, then we unhook this starting number and attach it to the beginning of second. After a finite number of steps we have y subsequences, and their starting numbers will form an increasing subsequence of length y. Since we assumed that $y > x$ we reached a contradiction.

Thus it follows that $y = x$.

Restoring the sequences: The desired partition of the sequence into subsequences can be done greedily. I.e. go from left to right and assign the current number or that subsequence ending with the minimal number which is greater than or equal to the current one.

Practice Problems

- ACMSGURU - "North-East"
- Codeforces - LCIS
- Codeforces - Tourist
- SPOJ - DOSA
- SPOJ - HMLIS
- SPOJ - ONEXLIS
- SPOJ - SUPPER
- Topcoder - AutoMarket
- Topcoder - BridgeArrangement
- Topcoder - IntegerSequence
- UVA - Back To Edit Distance
- UVA - Happy Birthday
- UVA - Tiling Up Blocks

Contributors:
jakobkogler (92.22%) sgtlaugh (3.33%) adamant-pwn (1.67%) Sergey-A-Dovgal (1.67%) JustAnAverageGuy (0.28%) Nil-Cipher (0.28%) wikku (0.28%) karan221 (0.28%)

Last update: June 8, 2022 Translated From: e-maxx.ru

Games on arbitrary graphs

Let a game be played by two players on an arbitrary graph G. I.e. the current state of the game is a certain vertex. The players perform moves by turns, and move from the current vertex to an adjacent vertex using a connecting edge. Depending on the game, the person that is unable to move will either lose or win the game.

We consider the most general case, the case of an arbitrary directed graph with cycles. It is our task to determine, given an initial state, who will win the game if both players play with optimal strategies or determine that the result of the game will be a draw.

We will solve this problem very efficiently. We will find the solution for all possible starting vertices of the graph in linear time with respect to the number of edges: $O(m)$.

Description of the algorithm

We will call a vertex a winning vertex, if the player starting at this state will win the game, if they play optimally (regardless of what turns the other player makes). Similarly, we will call a vertex a losing vertex, if the player starting at this vertex will lose the game, if the opponent plays optimally.

For some of the vertices of the graph, we already know in advance that they are winning or losing vertices: namely all vertices that have no outgoing edges.

Also we have the following **rules**:

- if a vertex has an outgoing edge that leads to a losing vertex, then the vertex itself is a winning vertex.
- if all outgoing edges of a certain vertex lead to winning vertices, then the vertex itself is a losing vertex.
- if at some point there are still undefined vertices, and neither will fit the first or the second rule, then each of these vertices, when used as a starting vertex, will lead to a draw if both player play optimally.

Thus, we can define an algorithm which runs in $O(nm)$ time immediately. We go through all vertices and try to apply the first or second rule, and repeat.

However, we can accelerate this procedure, and get the complexity down to $O(m)$.

We will go over all the vertices, for which we initially know if they are winning or losing states. For each of them, we start a depth first search. This DFS will move back over the reversed edges. First of all, it will not enter vertices which already are defined as winning or losing vertices. And further, if the search goes from a losing vertex to an undefined vertex, then we mark this one as a winning vertex, and continue the DFS using this new vertex. If we go from a winning vertex to an undefined vertex, then we must check whether all edges from this one leads to winning vertices. We can perform this test in $O(1)$ by storing the number of edges that lead to a winning vertex for each vertex. So if we go from a winning vertex to an undefined one, then we increase the counter, and check if this number is equal to the number of outgoing edges. If this is the case, we can mark this vertex as a losing vertex, and continue the DFS from this vertex. Otherwise we don't know yet, if this vertex is a winning or losing vertex, and therefore it doesn't make sense to keep continuing the DFS using it.

In total we visit every winning and every losing vertex exactly once (undefined vertices are not visited), and we go over each edge also at most one time. Hence the complexity is $O(m)$.

Implementation

Here is the implementation of such a DFS. We assume that the variable `adj_rev` stores the adjacency list for the graph in **reversed** form, i.e. instead of storing the edge (i, j) of the graph, we store (j, i). Also for each vertex we assume that the outgoing degree is already computed.

```
vector<vector<int>> adj_rev;

vector<bool> winning;
vector<bool> losing;
vector<bool> visited;
vector<int> degree;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj_rev[v]) {
        if (!visited[u]) {
            if (losing[v])
                winning[u] = true;
            else if (--degree[u] == 0)
                losing[u] = true;
            else
                continue;
            dfs(u);
        }
    }
}
```

Example: "Policeman and thief"

Here is a concrete example of such a game.

There is $m \times n$ board. Some of the cells cannot be entered. The initial coordinates of the police officer and of the thief are known. One of the cells is the exit. If the policeman and the thief are located at the same cell at any moment, the policeman wins. If the thief is at the exit cell (without the policeman also being on the cell), then the thief wins. The policeman can walk in all 8 directions, the thief only in 4 (along the coordinate axis). Both the policeman and the thief will take turns moving. However they also can skip a turn if they want to. The first move is made by the policeman.

We will now **construct the graph**. For this we must formalize the rules of the game. The current state of the game is determined by the coordinates of the police offices P, the coordinates of the thief T, and also by whose turn it is, let's call this variable P_{turn} (which is true when it is the policeman's turn). Therefore a vertex of the graph is determined by the triple (P, T, P_{turn}) The graph then can be easily constructed, simply by following the rules of the game.

Next we need to determine which vertices are winning and which are losing ones initially. There is a **subtle point** here. The winning / losing vertices depend, in addition to the coordinates, also on P_{turn} - whose turn it. If it is the policeman's turn, then the vertex is a winning vertex, if the coordinates of the policeman and the thief coincide, and the vertex is a losing one if it is not a winning one and the thief is on the exit vertex. If it is the thief's turn, then a vertex is a losing vertex, if the coordinates of the two players coincide, and it is a winning vertex if it is not a losing one, and the thief is at the exit vertex.

The only point before implementing is not, that you need to decide if you want to build the graph **explicitly** or just construct it **on the fly**. On one hand, building the graph explicitly will be a lot easier and there is less chance of making mistakes. On the other hand, it will increase the amount of code and the running time will be slower than if you build the graph on the fly.

The following implementation will construct the graph explicitly:

```
struct State {
    int P, T;
    bool Pstep;
};

vector<State> adj_rev[100][100][2]; // [P][T][Pstep]
bool winning[100][100][2];
bool losing[100][100][2];
bool visited[100][100][2];
int degree[100][100][2];
```



```
void dfs(State v) {
    visited[v.P][v.T][v.Pstep] = true;
    for (State u : adj_rev[v.P][v.T][v.Pstep]) {
        if (!visited[u.P][u.T][u.Pstep]) {
            if (losing[v.P][v.T][v.Pstep])
                winning[u.P][u.T][u.Pstep] = true;
            else if (--degree[u.P][u.T][u.Pstep] == 0)
                losing[u.P][u.T][u.Pstep] = true;
            else
                continue;
            dfs(u);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> a(n);
    for (int i = 0; i < n; i++)
        cin >> a[i];

    for (int P = 0; P < n*m; P++) {
        for (int T = 0; T < n*m; T++) {
            for (int Pstep = 0; Pstep <= 1; Pstep++) {
                int Px = P/m, Py = P%m, Tx = T/m, Ty = T%m;
                if (a[Px][Py]=='*' || a[Tx][Ty]=='*')
                    continue;

                bool& win = winning[P][T][Pstep];
                bool& lose = losing[P][T][Pstep];
                if (Pstep) {
                    win = Px==Tx && Py==Ty;
                    lose = !win && a[Tx][Ty] == 'E';
                } else {
                    lose = Px==Tx && Py==Ty;
                    win = !lose && a[Tx][Ty] == 'E';
                }
                if (win || lose)
                    continue;

                State st = {P,T,!Pstep};
                adj_rev[P][T][Pstep].push_back(st);
                st.Pstep = Pstep;
                degree[P][T][Pstep]++;

                const int dx[] = {-1, 0, 1, 0, -1, -1, 1, 1};
                const int dy[] = {0, 1, 0, -1, -1, 1, -1, 1};
                for (int d = 0; d < (Pstep ? 8 : 4); d++) {
                    int PPx = Px, PPy = Py, TTx = Tx, TTy = Ty;
                    if (Pstep) {
                        PPx += dx[d];
                        PPy += dy[d];
```

```
                    } else {
                        TTx += dx[d];
                        TTy += dy[d];
                    }

                    if (PPx >= 0 && PPx < n && PPy >= 0 && PPy < m && a[PPx][PPy]
!= '*' &&
                        TTx >= 0 && TTx < n && TTy >= 0 && TTy < m && a[TTx][TTy]
!= '*')
                    {
                        adj_rev[PPx*m+PPy][TTx*m+TTy][!Pstep].push_back(st);
                        ++degree[P][T][Pstep];
                    }
                }
            }
        }
    }

    for (int P = 0; P < n*m; P++) {
        for (int T = 0; T < n*m; T++) {
            for (int Pstep = 0; Pstep <= 1; Pstep++) {
                if ((winning[P][T][Pstep] || losing[P][T][Pstep]) && !visited[P]
[T][Pstep])
                    dfs({P, T, (bool)Pstep});
            }
        }
    }

    int P_st, T_st;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (a[i][j] == 'P')
                P_st = i*m+j;
            else if (a[i][j] == 'T')
                T_st = i*m+j;
        }
    }

    if (winning[P_st][T_st][true]) {
        cout << "Police catches the thief"  << endl;
    } else if (losing[P_st][T_st][true]) {
        cout << "The thief escapes" << endl;
    } else {
        cout << "Draw" << endl;
    }
}
```

Contributors:

jakobkogler (94.17%) AbhijeetKrishnan (3.14%) adamant-pwn (2.69%)

Last update: June 2, 2023 Translated From: e-maxx.ru

Search for connected components in a graph

Given an undirected graph G with n nodes and m edges. We are required to find in it all the connected components, i.e, several groups of vertices such that within a group each vertex can be reached from another and no path exists between different groups.

An algorithm for solving the problem

- To solve the problem, we can use Depth First Search or Breadth First Search.
- In fact, we will be doing a series of rounds of DFS: The first round will start from first node and all the nodes in the first connected component will be traversed (found). Then we find the first unvisited node of the remaining nodes, and run Depth First Search on it, thus finding a second connected component. And so on, until all the nodes are visited.
- The total asymptotic running time of this algorithm is $O(n + m)$: In fact, this algorithm will not run on the same vertex twice, which means that each edge will be seen exactly two times (at one end and at the other end).

Implementation

```
int n;
vector<vector<int>> adj;
vector<bool> used;
vector<int> comp;

void dfs(int v) {
    used[v] = true ;
    comp.push_back(v);
    for (int u : adj[v]) {
        if (!used[u])
            dfs(u);
    }
}

void find_comps() {
    fill(used.begin(), used.end(), 0);
    for (int v = 0; v < n; ++v) {
        if (!used[v]) {
            comp.clear();
            dfs(v);
            cout << "Component:" ;
            for (int u : comp)
                cout << ' ' << u;
            cout << endl ;
        }
    }
}
```

- The most important function that is used is `find_comps()` which finds and displays connected components of the graph.
- The graph is stored in adjacency list representation, i.e `adj[v]` contains a list of vertices that have edges from the vertex `v`.
- Vector `comp` contains a list of nodes in the current connected component.

Iterative implementation of the code

Deeply recursive functions are in general bad. Every single recursive call will require a little bit of memory in the stack, and per default programs only have a limited amount of stack space. So when you do a recursive DFS over a connected graph with millions of nodes, you might run into stack overflows.

It is always possible to translate a recursive program into an iterative program, by manually maintaining a stack data structure. Since this data structure is allocated on the heap, no stack overflow will occur.

```
int n;
vector<vector<int>> adj;
vector<bool> used;
vector<int> comp;

void dfs(int v) {
    stack<int> st;
    st.push(v);

    while (!st.empty()) {
        int curr = st.top();
        st.pop();
        if (!used[curr]) {
            used[curr] = true;
            comp.push_back(curr);
            for (int i = adj[curr].size() - 1; i >= 0; i--) {
                st.push(adj[curr][i]);
            }
        }
    }
}

void find_comps() {
    fill(used.begin(), used.end(), 0);
    for (int v = 0; v < n ; ++v) {
        if (!used[v]) {
            comp.clear();
            dfs(v);
            cout << "Component:" ;
            for (int u : comp)
                cout << ' ' << u;
            cout << endl ;
        }
    }
}
```

Practice Problems

- SPOJ: CCOMPS
- SPOJ: CT23E
- CODECHEF: GERALD07
- CSES : Building Roads

Contributors:
jakobkogler (35.19%) pratikkumar399 (25.0%) gabrielsimoes (16.67%) saurabh0402 (10.19%) adamant-pwn (5.56%) sunil-sangwan (3.7%) nhuhoang0701 (1.85%) vedant-z (0.93%) RodionGork (0.93%)

Last update: October 16, 2023 Translated From: e-maxx.ru

Chinese Remainder Theorem

The Chinese Remainder Theorem (which will be referred to as CRT in the rest of this article) was discovered by Chinese mathematician Sun Zi.

Formulation

Let $m = m_1 \cdot m_2 \cdots m_k$, where m_i are pairwise coprime. In addition to m_i, we are also given a system of congruences

$$\begin{cases} a &\equiv a_1 \pmod{m_1} \\ a &\equiv a_2 \pmod{m_2} \\ &\vdots \\ a &\equiv a_k \pmod{m_k} \end{cases}$$

where a_i are some given constants. The original form of CRT then states that the given system of congruences always has *one and exactly one* solution modulo m.

E.g. the system of congruences

$$\begin{cases} a &\equiv 2 \pmod{3} \\ a &\equiv 3 \pmod{5} \\ a &\equiv 2 \pmod{7} \end{cases}$$

has the solution 23 modulo 105, because $23 \bmod 3 = 2$, $23 \bmod 5 = 3$, and $23 \bmod 7 = 2$. We can write down every solution as $23 + 105 \cdot k$ for $k \in \mathbb{Z}$.

Corollary

A consequence of the CRT is that the equation

$$x \equiv a \pmod{m}$$

is equivalent to the system of equations

$$\begin{cases} x &\equiv a_1 \pmod{m_1} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{cases}$$

(As above, assume that $m = m_1 m_2 \cdots m_k$ and m_i are pairwise coprime).

Solution for Two Moduli

Consider a system of two equations for coprime m_1, m_2:

$$\begin{cases} a \equiv a_1 \pmod{m_1} \\ a \equiv a_2 \pmod{m_2} \end{cases}$$

We want to find a solution for $a \pmod{m_1 m_2}$. Using the Extended Euclidean Algorithm we can find Bézout coefficients n_1, n_2 such that

$$n_1 m_1 + n_2 m_2 = 1.$$

In fact n_1 and n_2 are just the modular inverses of m_1 and m_2 modulo m_2 and m_1. We have $n_1 m_1 \equiv 1 \pmod{m_2}$ so $n_1 \equiv m_1^{-1} \pmod{m_2}$, and vice versa $n_2 \equiv m_2^{-1} \pmod{m_1}$.

With those two coefficients we can define a solution:

$$a = a_1 n_2 m_2 + a_2 n_1 m_1 \bmod m_1 m_2$$

It's easy to verify that this is indeed a solution by computing $a \bmod m_1$ and $a \bmod m_2$.

$$\begin{aligned} a &\equiv a_1 n_2 m_2 + a_2 n_1 m_1 &\pmod{m_1} \\ &\equiv a_1 (1 - n_1 m_1) + a_2 n_1 m_1 &\pmod{m_1} \\ &\equiv a_1 - a_1 n_1 m_1 + a_2 n_1 m_1 &\pmod{m_1} \\ &\equiv a_1 &\pmod{m_1} \end{aligned}$$

Notice, that the Chinese Remainder Theorem also guarantees, that only 1 solution exists modulo $m_1 m_2$. This is also easy to prove.

Lets assume that you have two different solutions x and y. Because $x \equiv a_i \pmod{m_i}$ and $y \equiv a_i \pmod{m_i}$, it follows that $x - y \equiv 0 \pmod{m_i}$ and therefore $x - y \equiv 0 \pmod{m_1 m_2}$ or equivalently $x \equiv y \pmod{m_1 m_2}$. So x and y are actually the same solution.

Solution for General Case

Inductive Solution

As $m_1 m_2$ is coprime to m_3, we can inductively repeatedly apply the solution for two moduli for any number of moduli. First you compute $b_2 := a \pmod{m_1 m_2}$ using the first two congruences, then you can compute $b_3 := a \pmod{m_1 m_2 m_3}$ using the congruences $a \equiv b_2 \pmod{m_1 m_2}$ and $a \equiv a_3 \pmod{m_3}$, etc.

Direct Construction

A direct construction similar to Lagrange interpolation is possible.

Let $M_i := \prod_{i \neq j} m_j$, the product of all moduli but m_i, and N_i the modular inverses $N_i := M_i^{-1} \bmod m_i$. Then a solution to the system of congruences is:

$$a \equiv \sum_{i=1}^{k} a_i M_i N_i \pmod{m_1 m_2 \cdots m_k}$$

We can check this is indeed a solution, by computing $a \bmod m_i$ for all i. Because M_j is a multiple of m_i for $i \neq j$ we have

$$\begin{aligned} a &\equiv \textstyle\sum_{j=1}^{k} a_j M_j N_j \pmod{m_i} \\ &\equiv a_i M_i N_i \pmod{m_i} \\ &\equiv a_i M_i M_i^{-1} \pmod{m_i} \\ &\equiv a_i \pmod{m_i} \end{aligned}$$

Implementation

```
struct Congruence {
    long long a, m;
};

long long chinese_remainder_theorem(vector<Congruence> const& congruences) {
    long long M = 1;
    for (auto const& congruence : congruences) {
        M *= congruence.m;
    }

    long long solution = 0;
    for (auto const& congruence : congruences) {
        long long a_i = congruence.a;
        long long M_i = M / congruence.m;
        long long N_i = mod_inv(M_i, congruence.m);
        solution = (solution + a_i * M_i % M * N_i) % M;
    }
    return solution;
}
```

Solution for not coprime moduli

As mentioned, the algorithm above only works for coprime moduli $m_1, m_2, \dots m_k$.

In the not coprime case, a system of congruences has exactly one solution modulo $\operatorname{lcm}(m_1, m_2, \dots, m_k)$, or has no solution at all.

E.g. in the following system, the first congruence implies that the solution is odd, and the second congruence implies that the solution is even. It's not possible that a number is both odd and even, therefore there is clearly no solution.

$$\begin{cases} a \equiv 1 \pmod 4 \\ a \equiv 2 \pmod 6 \end{cases}$$

It is pretty simple to determine is a system has a solution. And if it has one, we can use the original algorithm to solve a slightly modified system of congruences.

A single congruence $a \equiv a_i \pmod{m_i}$ is equivalent to the system of congruences $a \equiv a_i \pmod{p_j^{n_j}}$ where $p_1^{n_1} p_2^{n_2} \cdots p_k^{n_k}$ is the prime factorization of m_i.

With this fact, we can modify the system of congruences into a system, that only has prime powers as moduli. E.g. the above system of congruences is equivalent to:

$$\begin{cases} a \equiv 1 \pmod 4 \\ a \equiv 2 \equiv 0 \pmod 2 \\ a \equiv 2 \pmod 3 \end{cases}$$

Because originally some moduli had common factors, we will get some congruences moduli based on the same prime, however possibly with different prime powers.

You can observe, that the congruence with the highest prime power modulus will be the strongest congruence of all congruences based on the same prime number. Either it will give a contradiction with some other congruence, or it will imply already all other congruences.

In our case, the first congruence $a \equiv 1 \pmod 4$ implies $a \equiv 1 \pmod 2$, and therefore contradicts the second congruence $a \equiv 0 \pmod 2$. Therefore this system of congruences has no solution.

If there are no contradictions, then the system of equation has a solution. We can ignore all congruences except the ones with the highest prime power moduli. These moduli are now coprime, and therefore we can solve this one with the algorithm discussed in the sections above.

E.g. the following system has a solution modulo $\operatorname{lcm}(10, 12) = 60$.

$$\begin{cases} a \equiv 3 \pmod{10} \\ a \equiv 5 \pmod{12} \end{cases}$$

The system of congruence is equivalent to the system of congruences:

$$\begin{cases} a \equiv 3 \equiv 1 \pmod 2 \\ a \equiv 3 \equiv 3 \pmod 5 \\ a \equiv 5 \equiv 1 \pmod 4 \\ a \equiv 5 \equiv 2 \pmod 3 \end{cases}$$

The only congruence with same prime modulo are $a \equiv 1 \pmod 4$ and $a \equiv 1 \pmod 2$. The first one already implies the second one, so we can ignore the second one, and solve the following system with coprime moduli instead:

$$\begin{cases} a \equiv 3 \equiv 3 \pmod 5 \\ a \equiv 5 \equiv 1 \pmod 4 \\ a \equiv 5 \equiv 2 \pmod 3 \end{cases}$$

It has the solution $53 \pmod{60}$, and indeed $53 \bmod 10 = 3$ and $53 \bmod 12 = 5$.

Garner's Algorithm

Another consequence of the CRT is that we can represent big numbers using an array of small integers.

Instead of doing a lot of computations with very large numbers numbers, which might be expensive (think of doing divisions with 1000-digit numbers), you can pick a couple of coprime moduli and represent the large number as a system of congruences, and perform all operations on the system of equations. Any number a less than $m_1 m_2 \cdots m_k$ can be represented as an array $a_1, \ldots, a_k$, where $a \equiv a_i \pmod{m_i}$.

By using the above algorithm, you can again reconstruct the large number whenever you need it.

Alternatively you can represent the number in the **mixed radix** representation:

$$a = x_1 + x_2 m_1 + x_3 m_1 m_2 + \ldots + x_k m_1 \cdots m_{k-1} \text{ with } x_i \in [0, m_i)$$

Garner's algorithm, which is discussed in the dedicated article Garner's algorithm, computes the coefficients x_i. And with those coefficients you can restore the full number.

Practice Problems:

- Google Code Jam - Golf Gophers
- Hackerrank - Number of sequences
- Codeforces - Remainders Game

Contributors:
jakobkogler (74.78%) jxu (13.04%) algmyr (5.65%) adamant-pwn (3.04%) thanhtnguyen (1.74%) vatsalsharma376 (0.87%)
jatingaur18 (0.43%) MayankPratap (0.43%)

Last update: February 22, 2024 Translated From: e-maxx.ru

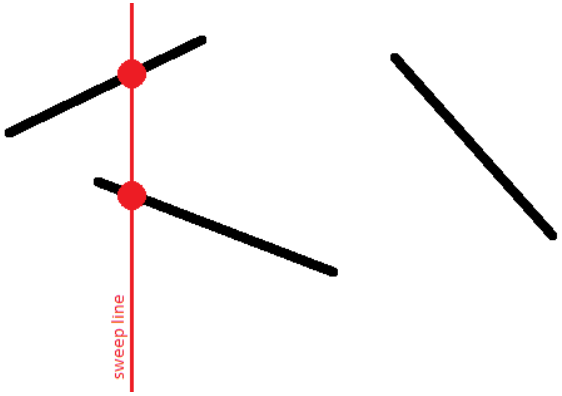
Search for a pair of intersecting segments

Given n line segments on the plane. It is required to check whether at least two of them intersect with each other. If the answer is yes, then print this pair of intersecting segments; it is enough to choose any of them among several answers.

The naive solution algorithm is to iterate over all pairs of segments in $O(n^2)$ and check for each pair whether they intersect or not. This article describes an algorithm with the runtime time $O(n \log n)$, which is based on the **sweep line algorithm**.

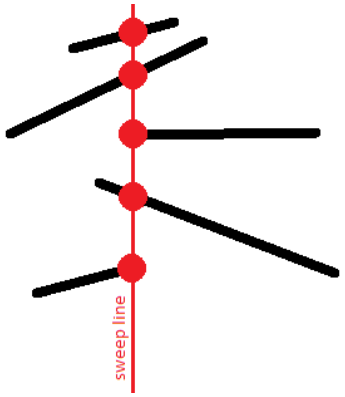
Algorithm

Let's draw a vertical line $x = -\infty$ mentally and start moving this line to the right. In the course of its movement, this line will meet with segments, and at each time a segment intersect with our line it intersects in exactly one point (we will assume that there are no vertical segments).

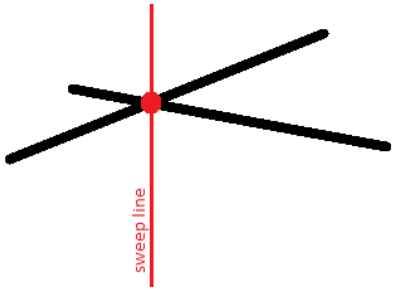


Thus, for each segment, at some point in time, its point will appear on the sweep line, then with the movement of the line, this point will move, and finally, at some point, the segment will disappear from the line.

We are interested in the **relative order of the segments** along the vertical. Namely, we will store a list of segments crossing the sweep line at a given time, where the segments will be sorted by their y-coordinate on the sweep line.



This order is interesting because intersecting segments will have the same y-coordinate at least at one time:



We formulate key statements:

- To find an intersecting pair, it is sufficient to consider **only adjacent segments** at each fixed position of the sweep line.
- It is enough to consider the sweep line not in all possible real positions $(-\infty \ldots +\infty)$, but **only in those positions when new segments appear or old ones disappear**. In other words, it is enough to limit yourself only to the positions equal to the abscissas of the end points of the segments.

- When a new line segment appears, it is enough to **insert** it to the desired location in the list obtained for the previous sweep line. We should only check for the intersection of the **added segment with its immediate neighbors in the list above and below**.
- If the segment disappears, it is enough to **remove** it from the current list. After that, it is necessary **check for the intersection of the upper and lower neighbors in the list**.
- Other changes in the sequence of segments in the list, except for those described, do not exist. No other intersection checks are required.

To understand the truth of these statements, the following remarks are sufficient:

- Two disjoint segments never change their **relative order**.
  In fact, if one segment was first higher than the other, and then became lower, then between these two moments there was an intersection of these two segments.
- Two non-intersecting segments also cannot have the same y-coordinates.
- From this it follows that at the moment of the segment appearance we can find the position for this segment in the queue, and we will not have to rearrange this segment in the queue any more: **its order relative to other segments in the queue will not change**.
- Two intersecting segments at the moment of their intersection point will be neighbors of each other in the queue.
- Therefore, for finding pairs of intersecting line segments is sufficient to check the intersection of all and only those pairs of segments that sometime during the movement of the sweep line at least once were neighbors to each other.
  It is easy to notice that it is enough only to check the added segment with its upper and lower neighbors, as well as when removing the segment — its upper and lower neighbors (which after removal will become neighbors of each other).
- It should be noted that at a fixed position of the sweep line, we must **first add all the segments** that start at this x-coordinate, and only **then remove all the segments** that end here.
  Thus, we do not miss the intersection of segments on the vertex: i.e. such cases when two segments have a common vertex.
- Note that **vertical segments** do not actually affect the correctness of the algorithm.
  These segments are distinguished by the fact that they appear and disappear at the same time. However, due to the previous comment, we know that all segments will be added to the queue first, and only then they will be deleted. Therefore, if the vertical segment intersects with some other segment opened at that moment (including the vertical one), it will be detected.
  In what place of the queue to place vertical segments? After all, a vertical segment does not have one specific y-coordinate, it extends for an entire segment along the y-coordinate. However, it is easy to understand that any coordinate from this segment can be taken as a y-coordinate.

Thus, the entire algorithm will perform no more than $2n$ tests on the intersection of a pair of segments, and will perform $O(n)$ operations with a queue of segments ($O(1)$ operations at the time of appearance and disappearance of each segment).

The final **asymptotic behavior of the algorithm** is thus $O(n \log n)$.

Implementation

We present the full implementation of the described algorithm:

```
const double EPS = 1E-9;

struct pt {
    double x, y;
};

struct seg {
    pt p, q;
    int id;

    double get_y(double x) const {
        if (abs(p.x - q.x) < EPS)
            return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
    }
};

bool intersect1d(double l1, double r1, double l2, double r2) {
    if (l1 > r1)
        swap(l1, r1);
    if (l2 > r2)
        swap(l2, r2);
    return max(l1, l2) <= min(r1, r2) + EPS;
}

int vec(const pt& a, const pt& b, const pt& c) {
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    return abs(s) < EPS ? 0 : s > 0 ? +1 : -1;
}

bool intersect(const seg& a, const seg& b)
{
    return intersect1d(a.p.x, a.q.x, b.p.x, b.q.x) &&
           intersect1d(a.p.y, a.q.y, b.p.y, b.q.y) &&
           vec(a.p, a.q, b.p) * vec(a.p, a.q, b.q) <= 0 &&
           vec(b.p, b.q, a.p) * vec(b.p, b.q, a.q) <= 0;
}

bool operator<(const seg& a, const seg& b)
{
    double x = max(min(a.p.x, a.q.x), min(b.p.x, b.q.x));
```

```
    return a.get_y(x) < b.get_y(x) - EPS;
}

struct event {
    double x;
    int tp, id;

    event() {}
    event(double x, int tp, int id) : x(x), tp(tp), id(id) {}

    bool operator<(const event& e) const {
        if (abs(x - e.x) > EPS)
            return x < e.x;
        return tp > e.tp;
    }
};

set<seg> s;
vector<set<seg>::iterator> where;

set<seg>::iterator prev(set<seg>::iterator it) {
    return it == s.begin() ? s.end() : --it;
}

set<seg>::iterator next(set<seg>::iterator it) {
    return ++it;
}

pair<int, int> solve(const vector<seg>& a) {
    int n = (int)a.size();
    vector<event> e;
    for (int i = 0; i < n; ++i) {
        e.push_back(event(min(a[i].p.x, a[i].q.x), +1, i));
        e.push_back(event(max(a[i].p.x, a[i].q.x), -1, i));
    }
    sort(e.begin(), e.end());

    s.clear();
    where.resize(a.size());
    for (size_t i = 0; i < e.size(); ++i) {
        int id = e[i].id;
        if (e[i].tp == +1) {
            set<seg>::iterator nxt = s.lower_bound(a[id]), prv = prev(nxt);
            if (nxt != s.end() && intersect(*nxt, a[id]))
                return make_pair(nxt->id, id);
            if (prv != s.end() && intersect(*prv, a[id]))
                return make_pair(prv->id, id);
            where[id] = s.insert(nxt, a[id]);
        } else {
            set<seg>::iterator nxt = next(where[id]), prv = prev(where[id]);
            if (nxt != s.end() && prv != s.end() && intersect(*nxt, *prv))
                return make_pair(prv->id, nxt->id);
            s.erase(where[id]);
        }
    }
```



```
    return make_pair(-1, -1);
}
```

The main function here is `solve()`, which returns the intersecting segments if exists, or $(-1, -1)$, if there are no intersections.

Checking for the intersection of two segments is carried out by the `intersect ()` function, using an **algorithm based on the oriented area of the triangle**.

The queue of segments is the global variable `s`, a `set<event>`. Iterators that specify the position of each segment in the queue (for convenient removal of segments from the queue) are stored in the global array `where`.

Two auxiliary functions `prev()` and `next()` are also introduced, which return iterators to the previous and next elements (or `end()`, if one does not exist).

The constant `EPS` denotes the error of comparing two real numbers (it is mainly used when checking two segments for intersection).

Problems

- TIMUS 1469 No Smoking!

Contributors:
singamandeep (92.05%) adamant-pwn (3.41%) jakobkogler (1.7%) SYury (1.7%) juan-c-s (0.57%) algmyr (0.57%)

Last update: December 7, 2023 Original

Knuth's Optimization

Knuth's optimization, also known as the Knuth-Yao Speedup, is a special case of dynamic programming on ranges, that can optimize the time complexity of solutions by a linear factor, from $O(n^3)$ for standard range DP to $O(n^2)$.

Conditions

The Speedup is applied for transitions of the form

$$dp(i,j) = \min_{i \le k < j} [dp(i,k) + dp(k+1,j) + C(i,j)].$$

Similar to divide and conquer DP, let $opt(i,j)$ be the value of k that minimizes the expression in the transition (opt is referred to as the "optimal splitting point" further in this article). The optimization requires that the following holds:

$$opt(i,j-1) \le opt(i,j) \le opt(i+1,j).$$

We can show that it is true when the cost function C satisfies the following conditions for $a \le b \le c \le d$:

1. $C(b,c) \le C(a,d)$;
2. $C(a,c) + C(b,d) \le C(a,d) + C(b,c)$ (the quadrangle inequality [QI]).

This result is proved further below.

Algorithm

Let's process the dp states in such a way that we calculate $dp(i,j-1)$ and $dp(i+1,j)$ before $dp(i,j)$, and in doing so we also calculate $opt(i,j-1)$ and $opt(i+1,j)$. Then for calculating $opt(i,j)$, instead of testing values of k from i to $j-1$, we only need to test from $opt(i,j-1)$ to $opt(i+1,j)$. To process (i,j) pairs in this order it is sufficient to use nested for loops in which i goes from the maximum value to the minimum one and j goes from $i+1$ to the maximum value.

Generic implementation

Though implementation varies, here's a fairly generic example. The structure of the code is almost identical to that of Range DP.

```
int solve() {
    int N;
    ... // read N and input
    int dp[N][N], opt[N][N];

    auto C = [&](int i, int j) {
        ... // Implement cost function C.
    };

    for (int i = 0; i < N; i++) {
        opt[i][i] = i;
        ... // Initialize dp[i][i] according to the problem
    }

    for (int i = N-2; i >= 0; i--) {
        for (int j = i+1; j < N; j++) {
            int mn = INT_MAX;
            int cost = C(i, j);
            for (int k = opt[i][j-1]; k <= min(j-1, opt[i+1][j]); k++) {
                if (mn >= dp[i][k] + dp[k+1][j] + cost) {
                    opt[i][j] = k;
                    mn = dp[i][k] + dp[k+1][j] + cost;
                }
            }
            dp[i][j] = mn;
        }
    }

    return dp[0][N-1];
}
```


Complexity

A complexity of the algorithm can be estimated as the following sum:

$$\sum_{i=1}^{N} \sum_{j=i+1}^{N} [opt(i+1,j) - opt(i,j-1)] = \sum_{i=1}^{N} \sum_{j=i}^{N-1} [opt(i+1,j+1) - opt(i,j)].$$

As you see, most of the terms in this expression cancel each other out, except for positive terms with $j = N$ and negative terms with $i = 1$. Thus, the whole sum can be estimated as

$$\sum_{k=1}^{N} [opt(k,N) - opt(1,k)] = O(n^2),$$

rather than $O(n^3)$ as it would be if we were using a regular range DP.

On practice

The most common application of Knuth's optimization is in Range DP, with the given transition. The only difficulty is in proving that the cost function satisfies the given conditions. The simplest case is when the cost function $C(i,j)$ is simply the sum of the elements of the subarray $S[i, i+1, \ldots, j]$ for some array (depending on the question). However, they can be more complicated at times.

Note that more than the conditions on the dp transition and the cost function, the key to this optimization is the inequality on the optimum splitting point. In some problems, such as the optimal binary search tree problem (which is, incidentally, the original problem for which this optimization was developed), the transitions and cost functions will be less obvious, however, one can still prove that $opt(i,j-1) \le opt(i,j) \le opt(i+1,j)$, and thus, use this optimization.

Proof of correctness

To prove the correctness of this algorithm in terms of $C(i,j)$ conditions, it suffices to prove that

$$opt(i,j-1) \le opt(i,j) \le opt(i+1,j)$$

assuming the given conditions are satisfied.

> **Lemma**
>
> $dp(i,j)$ also satisfies the quadrangle inequality, given the conditions of the problem are satisfied.

general heuristics here
...

Explore directions and write if they’re complicated (2 min)

Simplify the problem (1 min)

Play around with examples for simplified problem (2 min)

Carry out observations from (3) into playing around with orig. problem (2 min)

Carry out observations from (4) into more strict methods (3 min)

Debugging

Looking at code (and formulas) for typos after a 1 m break

Playing around with test cases

Greedy heuristics here
...

Specific to the 2020 regional problem I (Bitonic Array):

1) explore directions, notice if they're hard

2) simplify to sorted array rather than inc + dec

3) notice coord. line and BIT works, throw out sum |dist-dist'|, carry out that observation for playing around with orig. problem

4) suspect greedy vs. ds for orig. problem, but throw out brute forcing length of split because it overcomplicates things and that direction looks hard after exploring

5) go into greedy direction

6) from cheatsheet find some greedy heursitics that you prepared

one of them is below:

(General greedy heuristics)

1) consider each element independently when it's a problem about splits like this one (or subarrays, subsequences)

2) see if solve(array)=solve(array\specific element)+f is correct (greedy works)

3) try good specific elements, for permutations and sorting min/max is good

4) play around with test case by running through 1st specific element, 2nd specific element, see if choices are independent or if, less strongly, we can make them greedily - if so, we found a greedy direction

Last update: December 8, 2023 Original

Divide and Conquer DP

Divide and Conquer is a dynamic programming optimization.

Preconditions

Some dynamic programming problems have a recurrence of this form:

$$dp(i, j) = \min_{0 \le k \le j} dp(i-1, k-1) + C(k, j)$$

where $C(k, j)$ is a cost function and $dp(i, j) = 0$ when $j < 0$.

Say $0 \le i < m$ and $0 \le j < n$, and evaluating C takes $O(1)$ time. Then the straightforward evaluation of the above recurrence is $O(mn^2)$. There are $m \times n$ states, and n transitions for each state.

Let $opt(i, j)$ be the value of k that minimizes the above expression. Assuming that the cost function satisfies the quadrangle inequality, we can show that $opt(i, j) \le opt(i, j+1)$ for all i, j. This is known as the *monotonicity condition*. Then, we can apply divide and conquer DP. The optimal "splitting point" for a fixed i increases as j increases.

This lets us solve for all states more efficiently. Say we compute $opt(i, j)$ for some fixed i and j. Then for any $j' < j$ we know that $opt(i, j') \le opt(i, j)$. This means when computing $opt(i, j')$, we don't have to consider as many splitting points!

To minimize the runtime, we apply the idea behind divide and conquer. First, compute $opt(i, n/2)$. Then, compute $opt(i, n/4)$, knowing that it is less than or equal to $opt(i, n/2)$ and $opt(i, 3n/4)$ knowing that it is greater than or equal to $opt(i, n/2)$. By recursively keeping track of the lower and upper bounds on opt, we reach a $O(mn \log n)$ runtime. Each possible value of $opt(i, j)$ only appears in $\log n$ different nodes.

Note that it doesn't matter how "balanced" $opt(i, j)$ is. Across a fixed level, each value of k is used at most twice, and there are at most $\log n$ levels.

Generic implementation

Even though implementation varies based on problem, here's a fairly generic template. The function `compute` computes one row i of states `dp_cur`, given the previous row $i-1$ of states `dp_before`. It has to be called with `compute(0, n-1, 0, n-1)`. The function `solve` computes `m` rows and returns the result.

```
int m, n;
vector<long long> dp_before, dp_cur;

long long C(int i, int j);

// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr) {
    if (l > r)
        return;

    int mid = (l + r) >> 1;
    pair<long long, int> best = {LLONG_MAX, -1};

    for (int k = optl; k <= min(mid, optr); k++) {
        best = min(best, {(k ? dp_before[k - 1] : 0) + C(k, mid), k});
    }

    dp_cur[mid] = best.first;
    int opt = best.second;

    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
}

long long solve() {
    dp_before.assign(n,0);
    dp_cur.assign(n,0);

    for (int i = 0; i < n; i++)
        dp_before[i] = C(0, i);

    for (int i = 1; i < m; i++) {
        compute(0, n - 1, 0, n - 1);
        dp_before = dp_cur;
    }

    return dp_before[n - 1];
}
```

Things to look out for

The greatest difficulty with Divide and Conquer DP problems is proving the monotonicity of opt. One special case where this is true is when the cost function satisfies the quadrangle inequality, i.e., $C(a,c) + C(b,d) \leq C(a,d) + C(b,c)$ for all $a \leq b \leq c \leq d$. Many Divide and Conquer DP problems can also be solved with the Convex Hull trick or vice-versa. It is useful to know and understand both!

Practice Problems

- AtCoder - Yakiniku Restaurants
- CodeForces - Ciel and Gondolas (Be careful with I/O!)
- CodeForces - Levels And Regions
- CodeForces - Partition Game
- CodeForces - The Bakery
- CodeForces - Yet Another Minimization Problem
- Codechef - CHEFAOR
- CodeForces - GUARDS (This is the exact problem in this article.)
- Hackerrank - Guardians of the Lunatics
- Hackerrank - Mining
- Kattis - Money (ACM ICPC World Finals 2017)
- SPOJ - ADAMOLD
- SPOJ - LARMY
- SPOJ - NKLEAVES
- Timus - Bicolored Horses
- USACO - Circular Barn
- UVA - Arranging Heaps
- UVA - Naming Babies

References

- Quora Answer by Michael Levin
- Video Tutorial by "Sothe" the Algorithm Wolf

Contributors:

Last update: April 7, 2023 Original

Sparse Table

Sparse Table is a data structure, that allows answering range queries. It can answer most range queries in $O(\log n)$, but its true power is answering range minimum queries (or equivalent range maximum queries). For those queries it can compute the answer in $O(1)$ time.

The only drawback of this data structure is, that it can only be used on *immutable* arrays. This means, that the array cannot be changed between two queries. If any element in the array changes, the complete data structure has to be recomputed.

Intuition

Any non-negative number can be uniquely represented as a sum of decreasing powers of two. This is just a variant of the binary representation of a number. E.g. $13 = (1101)_2 = 8 + 4 + 1$. For a number x there can be at most $\lceil \log_2 x \rceil$ summands.

By the same reasoning any interval can be uniquely represented as a union of intervals with lengths that are decreasing powers of two. E.g. $[2, 14] = [2, 9] \cup [10, 13] \cup [14, 14]$, where the complete interval has length 13, and the individual intervals have the lengths 8, 4 and 1 respectively. And also here the union consists of at most $\lceil \log_2(\text{length of interval}) \rceil$ many intervals.

The main idea behind Sparse Tables is to precompute all answers for range queries with power of two length. Afterwards a different range query can be answered by splitting the range into ranges with power of two lengths, looking up the precomputed answers, and combining them to receive a complete answer.

Precomputation

We will use a 2-dimensional array for storing the answers to the precomputed queries. $\mathrm{st}[i][j]$ will store the answer for the range $[j, j + 2^i - 1]$ of length 2^i. The size of the 2-dimensional array will be $(K + 1) \times \mathrm{MAXN}$, where MAXN is the biggest possible array length. K has to satisfy $\mathrm{K} \geq \lfloor \log_2 \mathrm{MAXN} \rfloor$, because $2^{\lfloor \log_2 \mathrm{MAXN} \rfloor}$ is the biggest power of two range, that we have to support. For arrays with reasonable length ($\leq 10^7$ elements), $K = 25$ is a good value.

The MAXN dimension is second to allow (cache friendly) consecutive memory accesses.

```
int st[K + 1][MAXN];
```

Because the range $[j, j + 2^i - 1]$ of length 2^i splits nicely into the ranges $[j, j + 2^{i-1} - 1]$ and $[j + 2^{i-1}, j + 2^i - 1]$, both of length 2^{i-1}, we can generate the table efficiently using dynamic programming:

```
std::copy(array.begin(), array.end(), st[0]);

for (int i = 1; i <= K; i++)
    for (int j = 0; j + (1 << i) <= N; j++)
        st[i][j] = f(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
```

The function f will depend on the type of query. For range sum queries it will compute the sum, for range minimum queries it will compute the minimum.

The time complexity of the precomputation is $O(\mathrm{N} \log \mathrm{N})$.

Range Sum Queries

For this type of queries, we want to find the sum of all values in a range. Therefore the natural definition of the function f is $f(x, y) = x + y$. We can construct the data structure with:

```
long long st[K + 1][MAXN];

std::copy(array.begin(), array.end(), st[0]);

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

To answer the sum query for the range $[L, R]$, we iterate over all powers of two, starting from the biggest one. As soon as a power of two 2^i is smaller or equal to the length of the range ($= R - L + 1$), we process the first part of range $[L, L + 2^i - 1]$, and continue with the remaining range $[L + 2^i, R]$.

```
long long sum = 0;
for (int i = K; i >= 0; i--) {
    if ((1 << i) <= R - L + 1) {
        sum += st[i][L];
        L += 1 << i;
    }
}
```

Time complexity for a Range Sum Query is $O(K) = O(\log \mathrm{MAXN})$.

Range Minimum Queries (RMQ)

These are the queries where the Sparse Table shines. When computing the minimum of a range, it doesn't matter if we process a value in the range once or twice. Therefore instead of splitting a range into multiple ranges, we can also split the range into only two overlapping ranges with power of two length. E.g. we can split the range $[1, 6]$ into the ranges $[1, 4]$ and $[3, 6]$. The range minimum of $[1, 6]$ is clearly the same as the minimum of the range minimum of $[1, 4]$ and the range minimum of $[3, 6]$. So we can compute the minimum of the range $[L, R]$ with:

$$\min(\mathrm{st}[i][L], \mathrm{st}[i][R - 2^i + 1]) \quad \text{where } i = \log_2(R - L + 1)$$

This requires that we are able to compute $\log_2(R - L + 1)$ fast. You can accomplish that by precomputing all logarithms:

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

Alternatively, log can be computed on the fly in constant space and time:

```
// C++20
#include <bit>
int log2_floor(unsigned long i) {
    return std::bit_width(i) - 1;
}

// pre C++20
int log2_floor(unsigned long long i) {
    return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
}
```

This benchmark shows that using `lg` array is slower because of cache misses.

Afterwards we need to precompute the Sparse Table structure. This time we define f with $f(x, y) = \min(x, y)$.

```
int st[K + 1][MAXN];

std::copy(array.begin(), array.end(), st[0]);

for (int i = 1; i <= K; i++)
    for (int j = 0; j + (1 << i) <= N; j++)
        st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
```

And the minimum of a range $[L, R]$ can be computed with:

```
int i = lg[R - L + 1];
int minimum = min(st[i][L], st[i][R - (1 << i) + 1]);
```

Time complexity for a Range Minimum Query is $O(1)$.

Similar data structures supporting more types of queries

One of the main weakness of the $O(1)$ approach discussed in the previous section is, that this approach only supports queries of idempotent functions. I.e. it works great for range minimum queries, but it is not possible to answer range sum queries using this approach.

There are similar data structures that can handle any type of associative functions and answer range queries in $O(1)$. One of them is called Disjoint Sparse Table. Another one would be the Sqrt Tree.

Practice Problems

- SPOJ - RMQSQ
- SPOJ - THRBL
- Codechef - MSTICK
- Codechef - SEAD
- Codeforces - CGCDSSQ
- Codeforces - R2D2 and Droid Army
- Codeforces - Maximum of Maximums of Minimums
- SPOJ - Miraculous
- DevSkill - Multiplication Interval (archived)

Last update: June 8, 2022 Translated From: e-maxx.ru

Minimum stack / Minimum queue

In this article we will consider three problems: first we will modify a stack in a way that allows us to find the smallest element of the stack in $O(1)$, then we will do the same thing with a queue, and finally we will use these data structures to find the minimum in all subarrays of a fixed length in an array in $O(n)$

Stack modification

We want to modify the stack data structure in such a way, that it possible to find the smallest element in the stack in $O(1)$ time, while maintaining the same asymptotic behavior for adding and removing elements from the stack. Quick reminder, on a stack we only add and remove elements on one end.

To do this, we will not only store the elements in the stack, but we will store them in pairs: the element itself and the minimum in the stack starting from this element and below.

```
stack<pair<int, int>> st;
```

It is clear that finding the minimum in the whole stack consists only of looking at the value `stack.top().second`.

It is also obvious that adding or removing a new element to the stack can be done in constant time.

Implementation:

- Adding an element:

```
int new_min = st.empty() ? new_elem : min(new_elem, st.top().second);
st.push({new_elem, new_min});
```

- Removing an element:

```
int removed_element = st.top().first;
st.pop();
```

- Finding the minimum:

```
int minimum = st.top().second;
```

Queue modification (method 1)

Now we want to achieve the same operations with a queue, i.e. we want to add elements at the end and remove them from the front.

Here we consider a simple method for modifying a queue. It has a big disadvantage though, because the modified queue will actually not store all elements.

The key idea is to only store the items in the queue that are needed to determine the minimum. Namely we will keep the queue in nondecreasing order (i.e. the smallest value will be stored in the head), and of course not in any arbitrary way, the actual minimum has to be always contained in the queue. This way the smallest element will always be in the head of the queue. Before adding a new element to the queue, it is enough to make a "cut": we will remove all trailing elements of the queue that are larger than the new element, and afterwards add the new element to the queue. This way we don't break the order of the queue, and we will also not loose the current element if it is at any subsequent step the minimum. All the elements that we removed can never be a minimum itself, so this operation is allowed. When we want to extract an element from the head, it actually might not be there (because we removed it previously while adding a smaller element). Therefore when deleting an element from a queue we need to know the value of the element. If the head of the queue has the same value, we can safely remove it, otherwise we do nothing.

Consider the implementations of the above operations:

```
deque<int> q;
```

- Finding the minimum:

```
int minimum = q.front();
```

- Adding an element:

```
while (!q.empty() && q.back() > new_element)
    q.pop_back();
q.push_back(new_element);
```

- Removing an element:

```
if (!q.empty() && q.front() == remove_element)
    q.pop_front();
```

It is clear that on average all these operation only take $O(1)$ time (because every element can only be pushed and popped once).

Queue modification (method 2)

This is a modification of method 1. We want to be able to remove elements without knowing which element we have to remove. We can accomplish that by storing the index for each element in the queue. And we also remember how many elements we already have added and removed.

```
deque<pair<int, int>> q;
int cnt_added = 0;
int cnt_removed = 0;
```

- Finding the minimum:

```
int minimum = q.front().first;
```

- Adding an element:

```
while (!q.empty() && q.back().first > new_element)
    q.pop_back();
q.push_back({new_element, cnt_added});
cnt_added++;
```

- Removing an element:

```
if (!q.empty() && q.front().second == cnt_removed)
    q.pop_front();
cnt_removed++;
```

Queue modification (method 3)

Here we consider another way of modifying a queue to find the minimum in $O(1)$. This way is somewhat more complicated to implement, but this time we actually store all elements. And we also can remove an element from the front without knowing its value.

The idea is to reduce the problem to the problem of stacks, which was already solved by us. So we only need to learn how to simulate a queue using two stacks.

We make two stacks, `s1` and `s2`. Of course these stack will be of the modified form, so that we can find the minimum in $O(1)$. We will add new elements to the stack `s1`, and remove elements from the stack `s2`. If at any time the stack `s2` is empty, we move all elements from `s1` to `s2` (which essentially reverses the order of those elements). Finally finding the minimum in a queue involves just finding the minimum of both stacks.

Thus we perform all operations in $O(1)$ on average (each element will be once added to stack `s1`, once transferred to `s2`, and once popped from `s2`)

Implementation:

```
stack<pair<int, int>> s1, s2;
```

- Finding the minimum:

```
if (s1.empty() || s2.empty())
    minimum = s1.empty() ? s2.top().second : s1.top().second;
else
    minimum = min(s1.top().second, s2.top().second);
```

- Add element:

```
int minimum = s1.empty() ? new_element : min(new_element, s1.top().second);
s1.push({new_element, minimum});
```

- Removing an element:

```
if (s2.empty()) {
    while (!s1.empty()) {
        int element = s1.top().first;
        s1.pop();
        int minimum = s2.empty() ? element : min(element, s2.top().second);
        s2.push({element, minimum});
    }
}
int remove_element = s2.top().first;
s2.pop();
```

Finding the minimum for all subarrays of fixed length

Suppose we are given an array A of length N and a given $M \leq N$. We have to find the minimum of each subarray of length M in this array, i.e. we have to find:

$$\min_{0 \leq i \leq M-1} A[i], \min_{1 \leq i \leq M} A[i], \min_{2 \leq i \leq M+1} A[i], \ \ldots \ , \min_{N-M \leq i \leq N-1} A[i]$$

We have to solve this problem in linear time, i.e. $O(n)$.

We can use any of the three modified queues to solve the problem. The solutions should be clear: we add the first M element of the array, find and output its minimum, then add the next element to the queue and remove the first element of the array, find and output its minimum, etc. Since all operations with the queue are performed in constant time on average, the complexity of the whole algorithm will be $O(n)$.

Practice Problems

- Queries with Fixed Length
- Binary Land

Contributors:
jakobkogler (93.2%) adamant-pwn (3.14%) r7919 (1.57%) prashant-raghu (1.05%) saumya66 (0.52%) tanmay-sinha (0.52%)