SoC 2022 : Competitive Coding

Week-6

Greedy Algorithms

Contents

Greedy Algorithms

- As the name suggests, we make locally optimal solutions to come up with a globally optimal solution.
- Unlike dynamic programming, which solves the subproblems before making the first choice [e.g., knapsack or matrix-parenthesization], a greedy algorithm makes its first choice before solving any subproblems.
- But like DP, greedy algorithms also exploit the optimal substructure property.
- We need to think about / prove that a greedy choice at each step yields a globally optimal solution.

1. Activity Scheduling

We're given a set $S = \{a_1, \ldots, a_n\}$ of n proposed activities, and can do only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i, where $0 \le s_i < f_i$. Activities a_i and a_j are compatible if their intervals do not overlap.

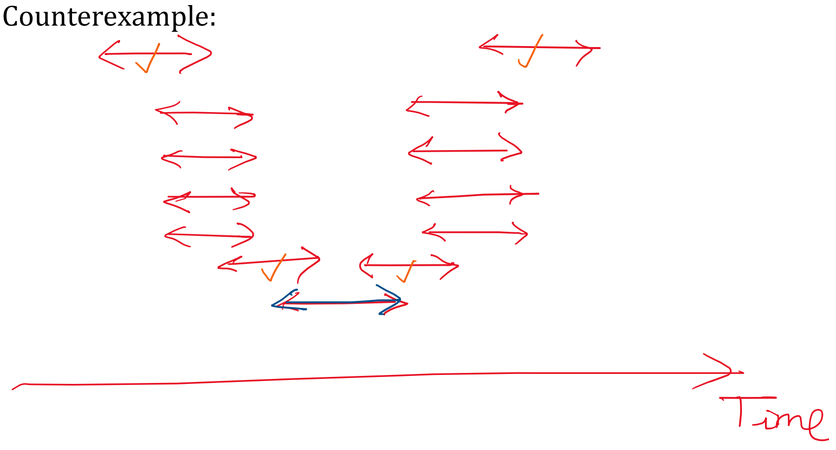
We wish to select a *maximum-size* subset of mutually compatible activities.

An exponential time algorithm is to iterate over all possible subsets.

Greedy Strategies that don't work [prove by counterexample]:

- Select activities greedily by smallest start time.
  Counterexample: A case where the first interval overlaps with everything else: {(0,10), (1,2), (3,4), (5,6)}.
  This strategy gives us {(0,10)}, but the optimal solution is {(1,2), (3,4), (5,6)}.

- Select an interval with the least number of overlaps with other intervals.
  Counterexample:



  The interval in blue has the least number of overlaps (2), but the optimal solution does not include it. The optimal solution is marked in orange.

- Select the shortest interval, and repeat greedily.
  Counterexample: A case where the shortest interval has many overlaps: {(1,5), (4,7), (6,11)}. The optimal solution is {(1,5), (6,11)}.

Correct greedy strategy:

Sort the intervals by finish times. Repeatedly pick the ones with smallest finish time.

Proof of optimality [CLRS]:
Let a_m be the activity with smallest finish time.
Let S' be an optimal solution. We will show that it is possible to create a set S'' that contains a_m and $|S''| = |S'|$.
Let the activity with smallest finish time in S' be a_j. If $a_j = a_m$, then we're done.
If not, then replacing a_j with a_m in S' will not affect compatibility of events in S'. This is because the start time of any activity other than a_j in S' is greater than the end time of a_j. Since end time of $a_m \leq$ that of a_j, the remark follows.
Thus, $S'' = (S' \cup \{a_m\}) \backslash \{a_j\}$ is the required set. Note that $|S''| = |S'|$, so S'' is also an optimal solution.

Could equivalently do by selecting activities greedily by "last start times". A similar proof works.

Running time: O(n log(n)) [sorting].

2. Activity Scheduling with Profit maximization

A modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a profit p_i. The objective is to maximize the total profits of the (non-overlapping) activities scheduled.

The above greedy approach won't work, since we might miss out on an interval with an extremely large profit.

Note that there was an optimal substructure in the previous problem [no profits version]:
- For any $i, j \in \{1, \dots, n\}$, let S_{ij} be the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Also let A_{ij} be the optimal solution formed from this subset.
- Suppose an event a_k is included in the solution. Then we're left with 2 subproblems: finding solutions for S_{ik} (set of activities starting after a_i finishes, and finishing before the start of a_k), and S_{kj}.
- We can see that A_{ij} must include optimal solutions for S_{ik} and S_{kj} by a simple contradiction argument. If this weren't the case, then we could include those optimal solutions for S_{ik} and/or S_{kj}, thus increasing $|A_{ij}|$. This contradicts the optimality of A_{ij}.
- Thus, $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$. [Optimal Substructure: the solution includes solutions of subproblems.]

This gives us a recurrence: Let $|A_{ij}| = c_{ij}$.
$$c_{ij} = \max_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\}$$
The max is over all activities in S_{ij}. If $S_{ij} = \phi$, then $c_{ij} = 0$ [base case].

We can use this recurrence to write a recursive solution. Note that memoization / DP matrix would greatly improve the time complexity, since c_{ij} would be needed multiple times.

Implementation Note: to directly improve a recursive solution (i.e., top-down), a C++ STL map/unordered_map can be used. In this case however, we know that the only keys would be in the space $\{1, \dots, n\} \times \{1, \dots, n\}$, so a matrix of size $O(n^2)$ would suffice as a memo [with worst-case O(1) access]. Also see this for unordered_map.

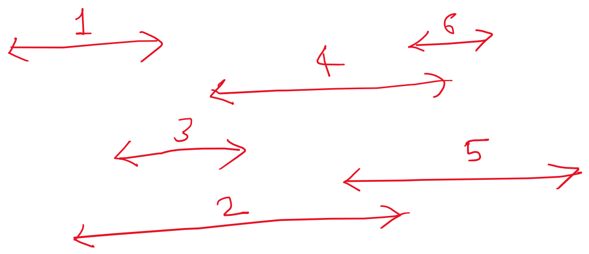
Now, we move on to the version with profits. In this case, the meaning of c_{ij} and the recurrence slightly change:
Let c_{ij} be the maximum profit achievable from S_{ij}.

$$c_{ij} = \max_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + p_k\}$$

A similar recursive/DP solution works.

3. Schedule all activities with minimum resources

We have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule **all** the activities using as few lecture halls as possible.



We've 6 activities above.

- Consider hall 1. We schedule activity 1 there.
- For activity 2, we need hall 2 [as there's an overlap].
- Hall 3 for activity 3.
- Hall 1 for activity 4.
- Hall 3 for activity 5.
- Hall 2 for activity 6.

Required #halls = 3.

Greedy strategy:
Sort activities by start time. For each activity, schedule it in the first empty lecture hall. If no such hall exists, add a new (unused till now) hall.

Proof of optimality:
The m^{th} hall is added for activity a only when there are activities going on in the earlier $m - 1$ halls at this point of time.
Thus, we have at least m overlapping activities. So, the required number of halls is at least m.

4. Jobs with deadlines and profits

We have an array of jobs. Each job has an associated deadline and profit, which we get only if the job is finished before the deadline. Also, each job takes a single unit of time. We need to maximize total profit, under the constraint that only one job can be scheduled at a given point in time.

Example

```
Job     Deadline  Profit
  a        2        100
  b        1        19
  c        2        27
  d        1        25
  e        3        15
Output: c, a, e
```

Greedy Strategy:

- Sort all jobs in decreasing order, according to profits. Iterate over these jobs as follows:
- For each job, place it in the rightmost available/empty slot, before its deadline.
- If no such slot exists, ignore this job.

Proof of optimality:

Let there be n jobs in total. Consider any solution S. Transform S as follows:
For job j_i in S, let d_i be its deadline, and $I_i \coloneqq [0, \min(n, d_i)]$. In this interval I_i, place j_i at the rightmost empty slot.
So, we've an optimal solution S with the property that for all $j_i \in S$, there is no idle time after j_i in the interval I_i.

Let S' be the output of the greedy algorithm. Note that S' also has the property that for all $j_i \in S'$, there is no idle time after j_i in the interval I_i [by the way we've designed the strategy].

Now, let j_k be the first job [by processing time] that occurs in S' but not in S.

If no such job exists, then $S' \subset S$. This is not possible: let j be the first job in S but not in S'. Since j is in $S \supset S'$, it is compatible with the other activities in S'. But while enumerating all jobs in the greedy algorithm, we would have included it [as it doesn't cause any deadline incompatibilities with existing jobs in S'].

Thus, there exists a job j_k such that $j_k \in S'$ and $j_k \notin S$. Let t be the time slot of j_k in S'.

If slot t were empty in S, then we would add j_k to that slot in S and then S won't be optimal. Contradiction.
So, t is not empty in S. Let this job be j_m. Note that j_m cannot have larger profit than j_k, otherwise the greedy strategy would have picked j_m instead. If it had lower profit, then again S isn't optimal.

Thus, we can swap j_m with j_k in the optimal solution. Proceeding in this way for all jobs ($\sim$ induction), we conclude that S' is optimal.

5. Fractional Knapsack

We've already seen this.

Knapsack, with relaxation of 0/1 condition. That is, any fraction of an object can also be taken.
Greedy strategy: For the object with highest 'price per unit weight', take this as much as possible [restricted by the bag capacity]. Repeat until the bag is full, or no more objects are left.

Proof of optimality:
Contradiction: Suppose an item with lesser 'price per unit weight' was taken, even though there was an item with larger ratio. Swap these two (in equal quantities), and the price increases. So, the earlier solution is not optimal.

6. Dijkstra's Algorithm for Single Source Shortest Paths

We've seen this greedy algorithm in the session on Graphs.
Key points:

- Works only for non-negative weights.
- Strategy is to maintain two sets, S and $V\backslash S$. Pick up the vertex in $V\backslash S$ that minimizes its distance from any vertex in S, and shift this vertex from $V\backslash S$ to S.
- Repeat until $V\backslash S$ is empty.
- For implementation, we used a priority-queue (min heap), and repeatedly inserted pairs of the form {vertex, current minimum distance from source} inside it.

7. Minimum Spanning Tree

Recall: A tree is a connected graph with no cycles. A spanning tree of a graph G is a subgraph which is a tree and which contains all vertices of G.

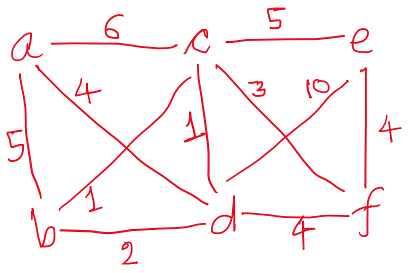
A minimum spanning tree (MST) for a weighted, connected, undirected graph G is a spanning tree with weight less than or equal to the weight of every other spanning tree of G.

Kruskal's Algorithm
- Sort all the edges in increasing order of their weights. Maintain a set of edges present in the output tree.
- Add the smallest weight edge to S, such that it does not form a cycle with the pre-existing edges in S.
- Repeat until no such edge remains.

For the second step, we would need to add edges, and then check if a cycle has been formed. We've seen how to check for a cycle in an undirected graph [using graph traversals].
This step can be sped up using the Union-Find data structure (aka Disjoint Set Union, or DSU). Will see that later.

Without any optimizations, time complexity is O(|V|.|E|) [sorting takes O(|E| log|E|), and note that $|E| \leq |V|^2 \Rightarrow \log|E| \leq 2\log|V|$].

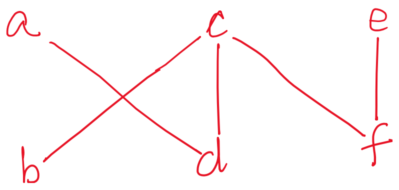
Example:



Edges sorted by weights (ascending):
bc, **cd**, bd, **cf**, df, **ef**, **ad**, ab, ce, ac, de. [edges in bold are included in the MST]

Note that we can break out of the execution once we have $|V| - 1$ edges.

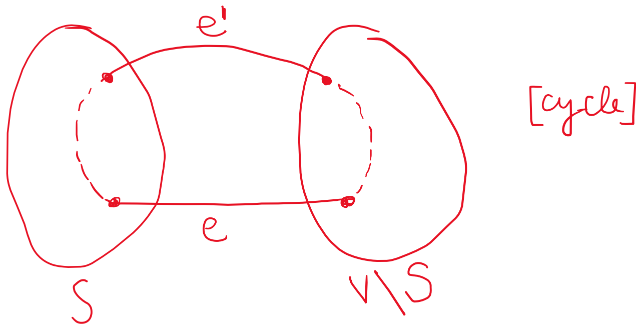
MST:



MST may not be unique. We could swap an edge with another compatible edge of the same weight. If all edge weights are distinct, then there is a unique MST: Proof.

Proof of optimality:

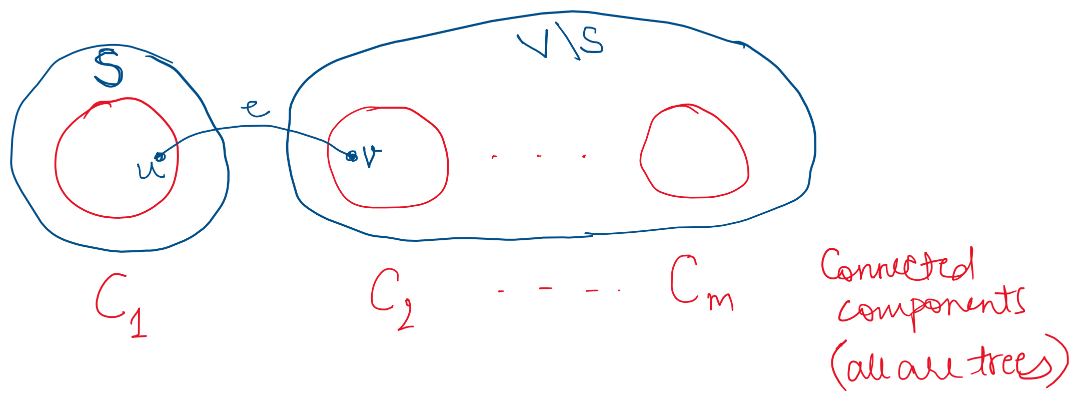
Claim: Let X be a subset of edges in an MST of G. Also let $S \subseteq V$ such that there is no $S \leftrightarrow V\backslash S$ edge in X. Let e be an edge of minimum weight between $S \leftrightarrow V\backslash S$.
Then, $X \cup \{e\}$ is part of some MST.



Proof: Let T be the MST containing X. If $e \in T$, done.
Else, $e \notin T$. Consider $T' = T \cup \{e\}$. This has exactly 1 cycle, since T is a tree. Moreover, this cycle must contain e [proof by contradiction].
Now, consider another edge e' in this cycle, such that it is across $S \leftrightarrow V\backslash S$. Note that such an edge (other than e) will always exist, since T is connected.
Define $T'' \coloneqq T' \cup \{e\} - \{e'\}$. This is a tree, and it spans G. Also, weight of $T'' \leq$ weight of T, since e is a minimum weight edge across $S \leftrightarrow V\backslash S$.
Since T is an MST, T'' also is an MST. Done.

Let T' be the output of Kruskal's algorithm. It can be proved that T' is spanning, connected, and acyclic [use contradiction].

To prove that T' has minimum weight:
<u>Claim:</u> At each step i of the algorithm, the set of selected edges E_i is part of some MST.



<u>Proof:</u> Induction on i. Base case can be shown using contradiction.
Induction step:
At step i, we have a forest [set of trees] formed using currently selected edges E_i. Next, the algorithm selects edge $e = (u, v)$ that the minimum weight of all unselected edges, such that addition of e doesn't create a cycle. That is, there is no $u \leftrightarrow v$ path using edges in E_i. So, we can define sets S and $V \backslash S$ such that there is no edge in E_i across $S \leftrightarrow V \backslash S$. By the above claim, $E_{i+1} = E_i \cup \{e\}$ is part of some MST.
Done.

Disjoint Set Union

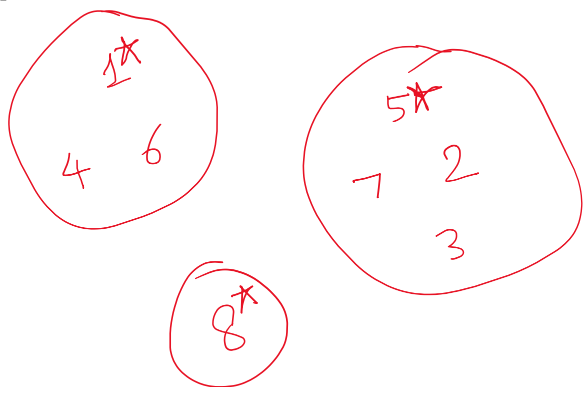
Also called Union-Find, because of its two main operations.

This data structure provides the following capabilities: We are given several sets of elements. 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.

Operations:

- make_set(v)   // initialization
- get(v)   // get the set that 'v' belongs to
- union_sets(a,b)   // merge the two sets that 'a' and 'b' belong to

Example:



The representative of each set is marked with an asterisk.
For example, get(6)=1, get(8)=8.

One way is to keep an array p[ ] (p for parent/representative), and many lists:

1	2	3	4	5	6	7	8
1	5	5	1	5	1	5	8

$1 \to 1,4,6$
$2 \to 2$
$3 \to 3$
$4 \to 4$
$5 \to 5,2,3,7$
$6 \to 6$
$7 \to 7$
$8 \to 8$

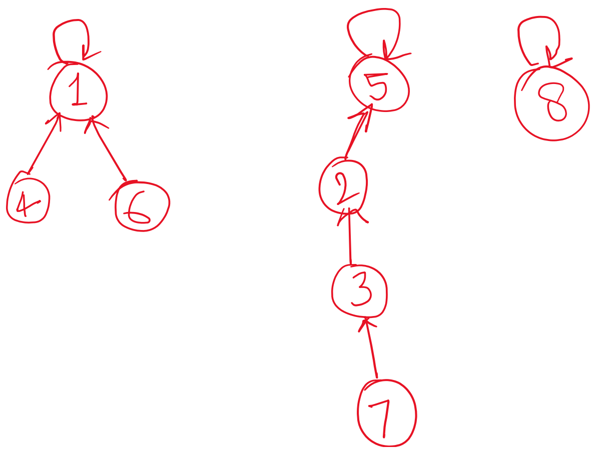
To do get(v), just return p[v] (O(1)).
To do a union:
For example, union(4,3). We first find their respective representatives 1, 5.

Then, for every element in the list of 5, mark its parent/representative = 1 in the array p[ ]. Note that this is the step for which we introduced the lists. Otherwise, we would have to traverse the entire array, searching and modifying all 5s.
Also modify the list for 1 (add elements 5, 2, 3, 7 to it). We can delete the list for 5 now, but it doesn't matter, since we've modified p[ ].

For union, we need to change a lot of values. This can be prevented by using a tree-like implementation. Now, with union, only a single node's parent is changed:

More Efficient Implementation:

Use trees:



The underlying structure is just an array parent[ ]:

i	1	2	3	4	5	6	7	8
parent[i]	1	5	2	1	5	1	3	8

The below time complexity analysis if for this tree implementation:

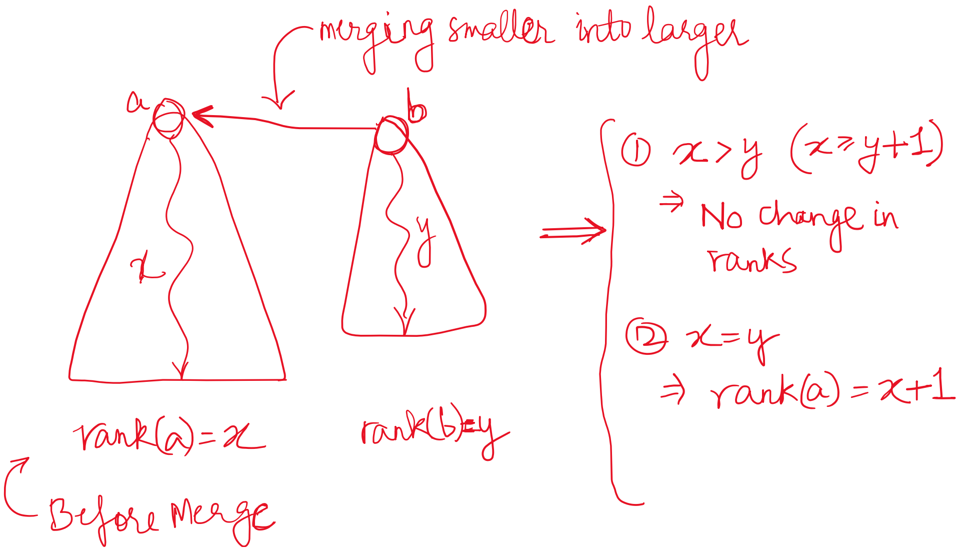
Note that get(v) can take O(n) in the worst case. Consider the example of having n sets $1, \ldots, n$. Then, we sequentially perform the following $n-1$ operations: union(1,2), union(1,3), ..., union(1,n).

This can be improved to O(log n) if we always merge the smaller list into the larger one. That is, make the new representative to be that one, which belongs to the larger list. This optimization is called the "Union by Size heuristic".

Another heuristic is union-by-rank: rank of a node := distance from farthest leaf in its subtree.

<u>Proof of logarithmic time complexity of get(), using union-by-size:</u>
Consider a smaller set (size m) being merged into a bigger one (size $\geq m$). The resulting set has a size $\geq 2m$. So, every <u>node</u> will be merged at most log(n) times, before the size of its set becomes n. Since every node is initialized with a single element, we can see that the distance of any node from its representative won't exceed log(n). Thus, get(v) is logarithmic in the number of nodes.
In the tree implementation [below], we can see that union consists of 2 get() calls, and some constant time operations. Thus, union is also bounded by the same complexity.

Using union-by-rank ['rank' defined above]:



Claim: Let $rank(a) = r$. Then, number of elements in the subtree of a are $\geq 2^r$.
Can be proved via induction. [Consider the 2 cases mentioned in the above diagram.]

Moreover, any trees having the same rank r are disjoint. Again, consider the 2 cases of union above.

Thus, for any rank r, there are at most $\frac{n}{2^r}$ subtrees that have rank r. So, at most 1 subtree can have rank $\log_2 n$. Thus, get() has a worst-case time complexity of $O(\log n)$. Since union ~ 2 get() calls + O(1), it is also bounded by the same.

Functions:

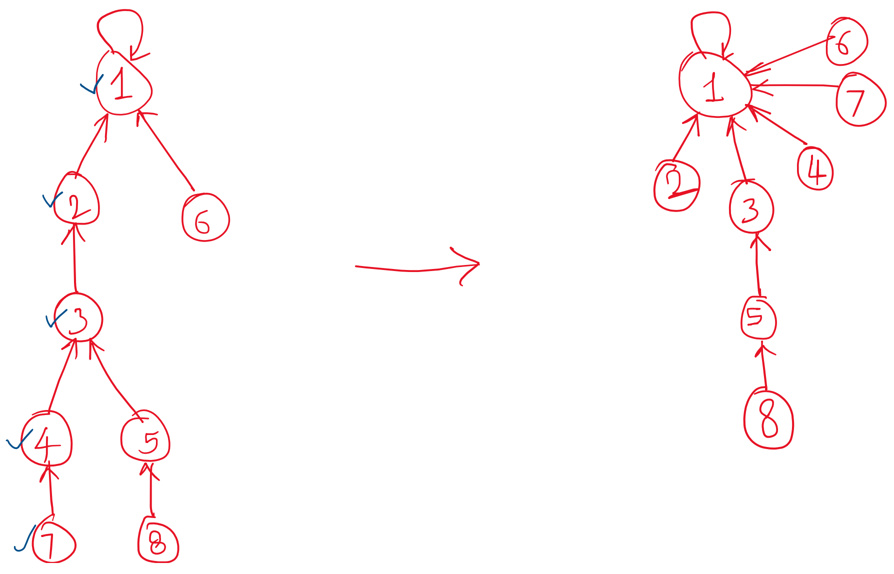
```
make_set(v):                    // single element set
  parent[v] = v
  size[v] = 1

get(v):
  while v!=parent[v]
    v=parent[v]
  return v

union(a,b):                     // union by size heuristic
  a = get(a)
  b = get(b)
  if (a != b):
    if (size[a] < size[b]):
      swap(a, b)
    parent[b] = a               // merge smaller into larger
    size[a] += size[b]
```

These heuristics still leave us with O(log n) complexity.

To improve on this, we use **Path Compression**:
Suppose we do: get(7). Then, 5 nodes [1,2,3,4,7] need to be traversed. To prevent this from happening the next time, we change the parent of 3,4,7 to 1.



```
get(v):                                    // recursive
  if (v == parent[v]):
    return v
  return parent[v] = get(parent[v])        // sets parent[v], and returns it
```

If we combine both optimizations – union by size/rank and path compression, we reach nearly constant time queries. The amortized time complexity is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. For a proof, refer to CLRS. For all practical purposes, $\alpha(n) \leq 4$. Ackermann function $A(m,n)$ grows very rapidly [$A(4,2) = 2^{65536} - 3$]. Its inverse grows very slowly.

Application to MST

The bottleneck in Kruskal's MST algorithm was to decide if adding an edge creates a cycle. We can use DSU as follows:
Maintain sets of connected vertices. Decision for an edge (u,v) can be taken, based on get(u) and get(v). If get(u)==get(v), addition of (u,v) introduces a cycle [since they are already connected].

See mst.cpp.

Other Examples

1. Jump

Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Your goal is to reach the last index in the minimum number of jumps, if possible.

Examples:
[2,3,1,1,4] – at least 2 jumps [2→3, 3→4] are required.
[1,0,0] – not possible.

DP/recursion+memo solution is O(n^2).

Repeatedly maintain the farthest position reachable.
For example, [2,3,1,1,4]: when we are at 2, the farthest to the right is the 2nd index.
If this farthest position doesn't increase in some iteration, return -1 [not possible].

See code: jump.cpp.

Problems

Will be shared.

Other Topics

- Prefix, suffix sums [and similarly, maxima, minima]
- Range queries [segment trees]: See 1, 2.
- GNU PBDS: See 1. Ordered set, with find_by_order() and order_of_key() in O(log n).