1 Disjoint Sets

As we’ve seen in Kruskal’s algorithm for finding MST’s, disjoint sets are useful data structures for maintaining sets that contain distinct elements. In this class, we represent them as trees, where the root of the tree is the canonical "name" of the tree.

Disjoint sets support the following operations:

(i) \textsc{makeSet}(x) — create a new set containing the single element $x$.

(ii) \textsc{union}(x, y) — replace sets containing $x$ and $y$ by their union.

(iii) \textsc{find}(x) — return name of set containing $x$.

We add for convenience the function \textsc{link}(x, y) where $x, y$ are roots: \textsc{link} changes the parent pointer of one of the roots to be the other root. In particular, \textsc{link}(\textsc{find}(x), \textsc{find}(y)) = \textsc{union}(x, y)$, so the main problem is to make the \textsc{find} operations efficient.

Union By Rank and Path Compression

To ensure $O(\log n)$ runtime of \textsc{find}, we try to ensure that the underlying tree structure is balanced under \textsc{union} operations. We do so by the union-by-rank heuristic which tries to preserve as low rank as possible by making the set with larger rank the parent.

We also have path compression, which is the idea that if we ever do a find on a node, we should update its parent to directly point at the root.

Exercise. \textit{Draw path compression after calling \textsc{find}(e).}

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{path_compression}
\caption{Path compression after calling \textsc{find}(e).}
\end{figure}

Solution.
The path up the tree is rewired as follows:
2 Divide and Conquer

Divide and Conquer algorithms work by recursively breaking the problem into smaller pieces and solving the subproblems.

Examples:

- Mergesort / StoogeSort
- Integer Multiplication & Matrix Multiplication (Strassen’s Algorithm)

Exercise. Given an array of $n$ elements, you want to determine whether there exists a majority element (that is an element which occurs at least $\lceil \frac{n+1}{2} \rceil$ times) and if so, output this element. Show how to do this in time $O(n \log n)$ using Divide and Conquer. By the way, can you do better?

Solution.

For $O(n \log n)$, obv. algorithm works. (run recursively on two halves, if both return "no majority" then return "no majority". If both return same majority element, return that element. If only one returns a majority element of if they return different elements, check if either of the returned elements is a majority by counting occurrences in combined array and return accordingly.)

There is a linear time solution. In the first pass of the list, we find an element that is the majority element if the majority element exists. Then we do a second pass to count the instances of that element and see if it is actually a majority element. The second pass is easy. For the first pass, maintain a counter and a "current candidate". Sweep along the list. At each element, if the counter is currently 0 then set the counter to 1 and "current candidate" to the element that we are looking at in the list. If the counter is non-zero then increment it or decrement it according to whether the current element in the list is or is not the "current candidate respectively.
3 Greedy Algorithms

A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice at every step. For some problems, a greedy algorithm will lead to a globally optimal solution; in other cases it will return an approximate solution.

Examples:

- Prim’s algorithm/Kruskal’s algorithm
- Horn-SAT
- Set Cover ($O(k \log n)$ approximation)

Consider a new example problem: a very simple scheduling problem:

Exercise. Suppose that we have a set $S = \{a_1, \ldots, a_n\}$ of proposed activities. Each activity $a_i$ has a start time $s_i$ and a finish time $f_i$. We can only run one activity at a time. Your job is to find a maximal set of compatible activities.

This problem exhibits a “subproblem structure”. If $a_m$ is included in a solution to this problem, then the solution needs to maximize the number of activities that occur before $a_m$ starts and after $a_m$ ends. In other words, an optimal solution to the whole problem requires optimal solutions to similar subproblems. This is called the optimal substructure property by CLRS and it underlies their discussion of dynamic programming. Greedy algorithms also rely on this property. A greedy algorithm chooses one activity to go in the solution-set and, in doing so, reduces the starting problem to one or more subproblems. It then recurses. Dynamic programming relies on finding optimal solutions to all possible subproblems and building up towards an optimal solution of the whole problem. A greedy algorithm uses clever insight to skip this process and instead plows downward, greedily forcing answers to the subproblems.

That’s a very complicated way of looking at greedy algorithms, but hopefully it sheds some light on dynamic programming, which is a much trickier topic. By the way, what’s a greedy solution to the above exercise?

Solution.
At each step, choose the job with earliest finish time. Recurse on the jobs that begin after it ends.

Here’s another exercise:

Exercise. Consider the problem of making change for $n$ cents using the fewest number of coins. Assume that each coin’s value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in the denominations that are powers of $c$, i.e., the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of $n$. 
Solution.
a. Choose the biggest coin that still fits. Suppose that an optimal solution exists that does not contain the biggest coin that we could have chosen at the first step. Do casework on the possible values of $n$. If $n < 5$ then we actually did use the biggest possible coin! If $5 \leq n < 10$ then by hypothesis we only used pennies. Replace five pennies with a nickel. etc.
b. An optimal solution may use at most $c - 1$ coins of any denomination. Assume that $n \geq c^m$ but we only used coins of denomination less than $c^m$. We have

$$
\sum_{i=0}^{m-1} a_i c^i \geq n
$$

but substituting the above constraint leads to a contradiction.
c. Use American coins: penny, dime, quarter. $n = 31$. 
