- note the chronological order of lecture content:
  - Module 1 (Basics and Graphs)
    - Week 1
      - 01 – Minimum Spanning Trees
    - Week 2
      - 02 – Kruskal's Algorithm
      - 03 – Abstract Data Types and Union Find
    - Week 3
      - 04 – Heaps
      - 05 – BFS & DFS
      - 06 – Graph Traversal Applications
    - Week 4
      - 07 – Dijkstra
      - 08 – A* Search
      - 09 – Bellman-Ford
    - Week 5
      - 10 – Asymptotics
      - 11 – Divide & Conquer: Closest Points
      - 12 – Divide & Conquer: Karatsuba-Strassen
  - Module 2 (Trees, Combinatorics, and Dynamic Programming)
    - Week 6
      - 13 – Binary Search Trees
      - 14 – Averaged Analysis
      - 15 – Skip Lists
      - 16 – Splay Trees
- an undirected graph is $G = (V, E)$, with V the vertices, E the edges. $V = \{V_1, V_2, ..., V_n\}$ is a set of objects, E a set of connections between the objects s.t. $\forall e \in E, e = \{u, v\}$ with $u, v \in V$.
- a directed graph differs in its edges E, where $\forall e \in E, e = (u, v)$ with $u, v \in V$.
- a subgraph of G is $H = (V_H, E_H)$ where $V_n \subseteq V, E_H \subseteq E$, and $(\forall e = (u, v) \in E)(u, v \in V_H)$
- connected component: maximal connected subgraph (can't connect more nodes)
- a cycle in $G = (V, E)$ is a sequence of nodes $\{v_k\} \in V$ s.t. $\{v_i, v_{i+1}\} \in E \land \{v_1, v_k\} \in E$
- a tree is an acyclic connected graph
- greedy algorithm:
  - local optimal decision to try solving a global problem
  - decisions are irrevocable (helpful for easy time complexity proofs)
- Minimum spanning tree (MST) problem: given an undirected connected graph G and nonnegative weights $d(e) = d(u, v)$, find the subgraph T that connects all vertices and minimizes

$$\text{cost}(T) = \sum_{\{u,v\} \in T} d(u, v)$$

  - think about why T must be a tree. if there are cycles, you can trivially rm an edge to cut costs
  - to make proofs simpler below, we assume the weights are unique. Then, to prove for cases where weights can be the same, we can just add small unique values. Apparently.
  - MST Cycle Property: let a cycle C be in G. the heaviest edge on the cycle C is not in G's MST.
  - MST Cut Property: let $S \subseteq V$, s.t. $1 \leq |S| < |V|$, i.e. S is not empty but not all nodes. Call a pair $(S, V \setminus S)$ a cut of the graph. Every MST of G contains the lightest edge on the cut (where an "edge on the cut" is some $e = \{u, v\}, u \in S, v \in V \setminus S$).
  - Prim's alg
    - tree growing paradigm
    - description
      - given graph G, choose arb node s, the start of T
      - repeating $|V| - 1$ times,
        - add to T, the lowest edge (and nodes) from "in T" to "not in T"
    - proof
      - The pair ("in T", "not in T") is a cut of G. By the Cut Property, the MST contains the lowest cost edge crossing this cut, which is by defn the next edge Prim's adds, i.e. Prim's adds edges in the MST.

        At any point in the algorithm, $T = (V_T, E_T)$ is a subgraph and a tree. T grows by 1 vertex and 1 edge at each step, stopping at $|V_G| - 1$ steps, and so results in a spanning tree. Since these edges are in the MST from above, the spanning tree is the MST. □
    - impl
      - (min) heap ADT
        - interface
          - `fn new(items) -> Self;`
          - `fn min() -> Item;`
          - `fn insert(item);`
          - `fn delete(item);`
        - heap-ordered tree (as an array) implementation

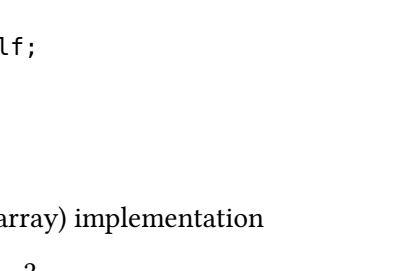

Figure 1: a heaparray. Notice the monotonic decreasing upward. Also notice the array implementation, where traverse_left$(i) = 2i$ and traverse_right$(i) = 2i + 1$.

          - `new` $\in O(n)$; take items as array, for all elements (indices right to left), sift down
          - `min` $\in O(1)$; it's at the top of the tree
          - `insert` $\in O(\log n)$; insert as leaf preserving shape (aka dense array), sift up
          - `delete` $\in O(\log n)$; swap with leaf, delete, sift swapped down
      - work of Prim's with a heap is $O(m \log n)$
        1. $O(n)$; create empty heaparray (for max n items)
        2. $O(1)$; choose arb start node u
        3. $O(\log n)$ work per heap operation, $O(m)$ times total (because we only nbors the edges that much, trust. see your example); $\forall v \in$ `nbors`(u), if v is closer to T than our saved distance from v to T, update this distance (and also set parent of v to u). With a heap, do `del(v); insert(v, d(u, v))` or just `decreaseKey(v, d(u, v))`. TODO parent??
        4. $O(\log n)$; set u to closest node to T (min of heap) and delete it
        5. $O(n)$ repetitions; while u exists, go to (3)
        - this is $O(m \log n + n \log n)$ from (3) and repeating (4), which is $\in O(m \log n)$
  - Reverse Delete
    - description
      - start with G
      - repeating $|V| - 1$ times
        - remove edges by decreasing weight unless it'd split the graph
    - proof
      - tree bc we removed all cycles by defn of algorithm
      - spanning bc $|V| - 1$ thing and non-split clause
      - minimum: at some step, let e be the next edge removed. Since it's removed, it's in a cycle and must be the largest edge in it. By cycle property, all removed edges are not in the MST.
  - Kruskal's alg
    - description
      - start with $T = (V, \{\})$, i.e. all the nodes and no edges
      - repeating $|V| - 1$ times
        - add edges in increasing order unless it creates a cycle
    - proof
      - tree bc we didn't connect in a cycle... but why is output connected?
      - spanning bc $|V| - 1$ thing and tree
      - minimum: at some step, let e be the next edge added. All rejected edges would have created cycles, and must be the max in the cycle they create by algo defn. This means, by cycle property, it's MST.
    - impl
      - union-find abstract data type (ADT)
        - interface
          - `fn new(items) -> Self;` splits items into that many sets
          - `fn find(item) -> Label;` finds the set the item belongs to
          - `fn union(a, b);` merges two sets
        - array-based implementation

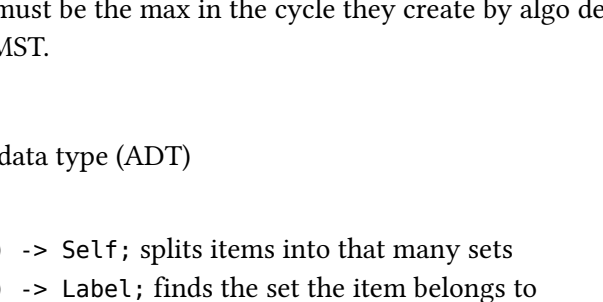

Figure 2: array-based union-find

          - `new` $\in O(n)$; creates the three lists
          - `find` $\in O(1)$; set array lookup
          - `union` $\in O(\log n)$ amortized;
            - procedure
              1. find smaller set ($x \leq y$)
              2. for each x_i, make `set[elem]` be y
              3. update sizes array
              4. prepend smaller to larger list
            - proof
              - (2) is computation. others $O(1)$
              - after k unions, $\leq 2k$ items touched
              - for any item v, `set[v]` is relabeled $\leq \log_2 2k$ times (TODO WHY) ($2k$ is the # of items in the largest set????)
              - $2k \log_2 2k \in O(k \log k)$ work
        - tree-based implementation

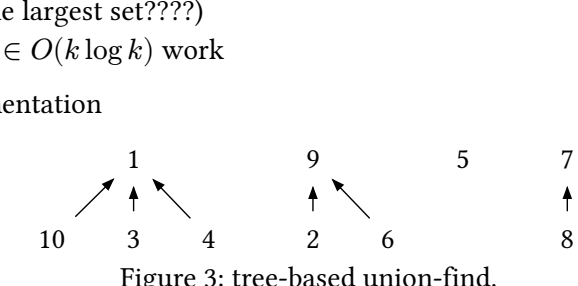
Figure 3: tree-based union-find.

          - `new` $\in O(n)$
            - let `struct Node { parent: Option<&Node>, height: uint }`
            - create an array of node addresses, so we can get from a node id to its node in $O(1)$
            - create all singleton nodes
          - `find` $\in O(\log n)$; follow ptrs to the root, which is the set label
            - proof
              - set at i is renamed at most $\log_2 n$ times because each renaming doubles size
              - depth of tree is number of renamings
          - `union` $\in O(1)$ (amortized?); move pointer, update height. is this really all we do?
      - work of Kruskal's for array-based union-find is $O(m \log n)$
        - sorting edges $\in O(m \log m) \in O(m \log n)$
        - because $m \leq n^2$, so $\log m \leq \log n^2 = 2 \log n$
        - at most $2m$ `find` operations $\in O(2m)$ (to check if edge would create cycle)
        - at most $n - 1$ `union` ops in $O(n \log n)$
        - total runtime $\in O(m \log n + 2m + n \log n) \in O(m \log n)$
      - work of Kruskal's for tree-based union-find is also $O(m \log n)$
        - sorting edges $\in O(m \log n)$ from above
        - at most $2m$ `find` ops $\in O(2m \log n)$
        - at most $n - 1$ union ops $\in O(n)$
        - total runtime $\in O(m \log n + 2m \log n + n) \in O(m \log n)$
- graph traversal problem: suppose a graph $G = (V, E)$. Find a path from a node s to t if it exists.
  - depth-first search: $O(n + m)$

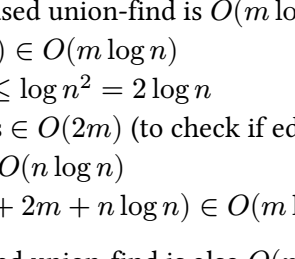
Figure 4: depth-first search

    - theorem: adjacent edges are not on the same level. That is, if $(x, y) \in E$, x is either an ancestor or descendant of y.
      - proof: WLOG, let x ancestor of y. When we pass x, we haven't seen y. All nodes between initially seeing x and leaving x are decendants of x. So, y must have been explored before leaving x, as a descendant.
  - breadth-first search: $O(n + m)$

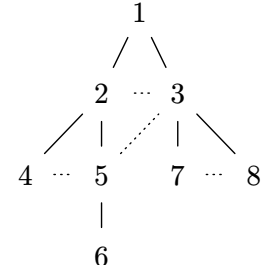
Figure 5: breadth-first search

    - theorem: adjacent edges are near the same level. That is, if $(x, y) \in E$, then $|\text{layer}(x) - \text{layer}(y)| \leq 1$
      - proof: 1) WLOG, AFSOC that layer$(x) <$ layer$(y) - 1$. 2) All nbors of x are added in or before layer$(x) + 1$. By 2, layer$(y) \leq$ layer$(x) + 1$. But by 1, layer$(y) >$ layer$(x) + 1$. $\rightarrow\leftarrow$.
  - these (and Prim's!) are specifc types of tree-growing algorithms

```
TreeGrowingAlg(graph G, initial node v, fn findnext)
    T = ({v}, {})
    S = set of nodes adj to v
    while S has items,
        e = nextEdge(G, S)
        T = T + e
        S = updateFrontier(G, S, e)
```

    - `next`/`update` is pop/push for DFS and de/enqueue for BFS, $O(m)$ across entire program.
      - same logic as Prim's
    - everything else in $O(n)$ (including loop)
    - thus, DFS and BFS are $\in O(n + m)$
  - is-bipartite problem
    - definition: a graph is bipartite if $\exists$ a two-coloring. that is, if you can divide the nodes into two colors s.t. all edges connect nodes of diff colors.

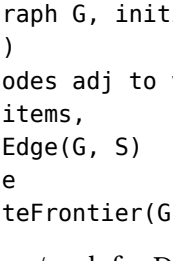
Figure 6: a bipartite graph

Figure 7: can't have odd cycles

    - solution: do a BFS from any node, swapping colors per layer. check if any edge is monolayer (since edges are between adj layers or same layer)
      - todo: prove correctness
- topological sort problem: given a DAG, assign an "order" to the nodes (a bijective $f : V \to [n]$).
  - Solution 1
    - Theorem: every DAG has a node with no incoming edges
      - think about it. what if there wasn't? then follow the edges backward. Since finite nodes in graph, eventually you return to already-seen: a cycle.
    - impl/time:

      initialize `num_incoming[w]` by counting edges with DFS in $O(n + m)$ and a "frontier" of 0-incoming nodes.

```
for i in 1..=n:
    find node u with no incoming (arb. in frontier)
    set f(u) = i
    del u from graph by updating its nbors' num_incoming and the frontier
```

      the loop happens n times. decrementing the neighbors' `num_incoming` happens m times overall. total algo time is $O(n + m)$.
    - proof: todo
  - Solution 2
    - do a DFS, track entering and leaving order. topological sort is the descending leaving number order.
    - this is clearly $O(n + m)$
    - proof: todo
- shortest path problem

Method	Time Bound	Notes
D/BFS	$O(n + m)$	unweighted only
Dijkstra	$O(m \log n)$	pos weights, source to all sinks
A*	$O(m \log n)$	needs an h (pref. admissable), source to one sink
Bellman-Ford	$O(nm)$	can process negative weights

Table 1: summary of options

  - as a note, running through these on paper is much better for comprehension than staring at the pseudocode
  - Dijkstra
    - use tree-growing paradigm, like Prim's
      - think about what this paradigm means for neg weights: if I choose a locally good weight, I might miss out on globally good (very negative) weights.
    - impl
      - define one start node s, tentative distances d (all ∞ except $d[s] = 0$), tentative parents p all `None`, frontier $F = V$.
      - while frontier has items
        - u = frontier node with min d
        - remove u from F (this "locks u in": its current d is the shortest possible.)
        - update d for neighbors of u where applicable
    - runtime
      - $O(m \log n)$, like Prim's and co.
      - create frontier as a heap in negligible time
      - every edge processed once, either do nothing or reducekey in heap for the $\log n$
    - proof
      - we prove that after considering n nodes, our tentative distances are perfect for those considered nodes
      - notice the base case holds
      - by induction,
        - consider the next node be added with edge (u, v)
        - let P_v be the path chosen by Dijkstra (smallest to frontier)
        - since the previously considered k node distances were correct, adding the smallest from the frontier works because we only expose one level of new distance info, which is all we need. We can take the shortest path because the distances are never negative, so later levels can never be better than what we know of this single next level. See diagram in the lecture notes.
  - A*
    - impl
      - literally just Dijkstra but with $h[u]$, a heuristic distance from u to dest t
      - that is, replace "min d" with "min $f = d + h$"
      - then, we can stop once we grow the dest t into our tree
    - an admissable h is one that always underestimates (or is equal to) the real u to t
      - if h admissable, A* will find the globally optimal solution (proof omitted)
  - Bellman-Ford
    - as mentioned above, tree-growing (Dijk/A*) doesn't work for negative weights. nor do they find (infinitely) negative weight cycles.
    - revised problem statement: find that $\exists$ negative cycle *or* shortest path
    - impl
      - again, use $d[s] = 0$, $d[\text{else}] = \infty$
      - relaxation (Ford) step: find any edge (u, v) s.t. $d[v] > d[u] + w(u, v)$. update d for that edge.
      - relax until can't anymore. if no cycle, this happens at most $n - 1$ times.
      - if we relaxed n times, there is a cycle (?)
- asymptotics
  - we want a formal definition of alg. efficiency for large problems
  - big O
    - $T(n) \in O(f(n))$ if $\exists n_0 \geq 0, c \geq 0, T(n) \leq cf(n) \forall n \geq n_0$
    - "exists some c linear multiplier such that $cf(n)$ dominates $T(n)$ after some n_0 large number."
  - big Ω
    - $T(n) \in \Omega(f(n))$ if $\exists \varepsilon \geq 0, n_0 \geq 0, T(n) \geq \varepsilon f(n) \forall n \geq n_0$
    - "exists some ε linear multiplier such that $\varepsilon f(n)$ is dominated by $T(n)$...
  - big Θ

    $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n)) \land T(n) \in \Omega(f(n))$
  - Theorem:

$$\exists c, \lim_{n \to \infty} \frac{T(n)}{g(n)} = c \iff T(n) \in \Theta(g(n))$$

    and similarly for the other two big complexities.
- divide and conquer
  - time complexity proofs
    - set up with old friend recurrence relations
    - tree Method
    - by induction (used to verify a known/given time bound)
      1. show $T(k) \leq f(k)$ for "small" k
      2. assume $T(k) \leq f(k)$ for $k < n$
      3. show $T(n) \in O(f(n))$ i.e. $T(n) \leq f(n)$, large n, blah blah blah
      - example: with $T(n) = 2T(\frac{n}{2}) + cn$, show $T(k) \leq 2ck \log 2k$ (i.e. $T(k) \in O(k \log k)$)
        - base case ($k = 2$): $2c \log 2$ uhhhh what.
    - by Master Theorem
      - given a recurrence of form

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^i), a \geq 1, b > 1$$

        - case 1: if $i < \log_b a$(i.e. $a > b^i$) $\longrightarrow T(n) = \Theta(n^{\log_b a})$
        - case 2: if $i = \log_b a$(i.e. $a = b^i$) $\longrightarrow T(n) = \Theta(n^i \log n)$
        - case 3: if $i > \log_b a$(i.e. $a < b^i$) $\longrightarrow T(n) = \Theta(n^i)$
      - think of it like "is the work at each leaf less, equal, or more than the work of splitting?" This decides what dominates.
      - unsimplified case 2 is $\Theta(n^i \log_b n)$ which matches the pattern better
  - mergesort
    - impl:

```
sort(L)
    if |L| == 2: return [min L, max L] in O(1)
    else
        L1 = sort(L[1 : |L|/2])
        L2 = sort(L[|L|/2 : |L|])
        ^ two calls to self on input length |L|/2
        return merge(L1, L2) in O(n)
```

    - time complexity
      - $T(n) = 2T(\frac{n}{2}) + c \cdot n$
      - Base case for $k = 2$. $c2 \log 2 = 2c \geq T(2)$
      - Induct.

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + c \cdot n \text{ by defn} \\ &\leq 2\left(c\frac{n}{2} \log \frac{n}{2}\right) + c \cdot n \text{ by IH} \\ &= cn \log n - cn \log_2 2 + cn \text{ by alg} \\ &= cn \log n \text{ by alg} \end{aligned}$$

      something something correct bounds of $O(n \log n)$
  - closest two points problem: given n points in 2d space, find closest two
    - divide step:
      - divide points (e.g. left and right side).
      - find closest in each side
    - merge step:
      - what if we cut the global closest distance?
      - call the smaller "closest distance" d
      - if that that global closest exists, it must at most d away from our split, a "possible region"
      - also, all pairs are at least d away from each other (both sides)
      - so, we just need to check all $O(n)$ points in the "possible region"
      - for each point, there is a small finite (15 lol) number of points that might be able to be the global closest
      - overall this means this merge step is $O(15n) \in O(n)$
    - this is $O(n \log n)$:D
  - (binary) integer multiplication problem
    - let's say we want $x \cdot y$, both n bits long
    - naïve is $\Theta(n^2)$ in length of binary number
    - we can get down to $\Theta(n^{1.58...})$ with Karatsuba's Alg!

    let $m = \frac{n}{2}$. then we can apply the "left shift" operation to write

$$\begin{aligned} x &= x_1 2^m + x_0 \\ y &= y_1 2^m + y_0 \end{aligned}$$

    Our multiplication becomes

$$\begin{aligned} x \cdot y &= (x_1 2^m + x_0)(y_1 2^m + y_0) \\ &= x_1 y_1 2^{2m} + (x_1 y_0 + x_0 y_1) 2^m + x_0 y_0 \end{aligned}$$

    Notice the terms

$$\begin{aligned} P_0 &= x_0 y_0 \\ P_1 &= x_1 y_1 \\ P_2 &= x_1 y_0 + x_0 y_1 \\ &= (x_1 + x_0)(y_1 + y_0) - P_0 - P_1 \end{aligned}$$

    for a total of only three unique multiplications! And recall they are half as wide as $x \cdot y$. Our original multiplication becomes

$$x \cdot y = ... = P_1 2^{2m} + P_2 2^m + P_0$$

    From this, we can write out our recurrence relation. Let n be some 2^k. Since we do those three multiplications on half the size, $T(2^k) = 3T(2^{k-1}) + c2^k$.

    The notes then do a lot of math nonsense. Surely the tree method is easier :(
  - matmul problem
    - naïve is $\Theta(n^3)$ in (square) matrix side length
    - with Strassen's alg, $\Theta(n^{2.807}...)$

    very similar of form to Karatuba's, using algebraic manipulations
- amortized and average analysis
  - skip lists

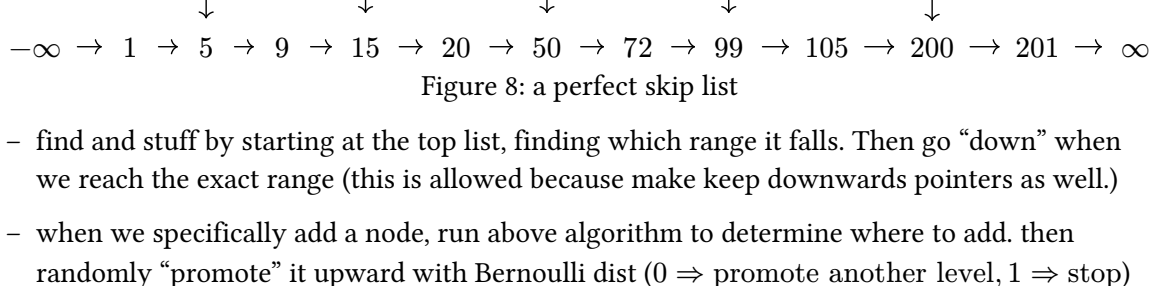
Figure 8: a perfect skip list

    - find and stuff by starting at the top list, finding which range it falls. Then go "down" when we reach the exact range (this is allowed because make keep downwards pointers as well.)
    - when we specifically add a node, run above algorithm to determine where to add. then randomly "promote" it upward with Bernoulli dist ($0 \Rightarrow$ promote another level, $1 \Rightarrow$ stop)
    - $O(\log n)$ all operations or whatever. consider an arb. search, traced backwards. Averaging means half ops go "up," half go "left." Expected value turns out to be $O(\log n)$ ops because getting to "top" is expected $2 \log n$ total skips, and we get to start from the top in $O(1)$.