CS 124 Homework 3: Spring 2024

Collaborators:

No. of late days used on previous psets:
No. of late days used after including this pset:

Homework is due Wednesday Feb 28 at 11:59pm ET. Please remember to select pages when you submit on gradescope. Each problem with incorrectly selected pages will lose 5 points.

Try to make your answers as clear and concise as possible; style may count in your grades. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan your papers to turn them in.

Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential-time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or no credit. Again, try to make your answers clear and concise.

Problems

1. **Detecting whether an edge lies in an MST (10 points)**

   (a) **(5 points)** Let G be a weighted graph in which all edge weights are distinct. Prove that an edge e of a graph G belongs in some MST of G if and only if the following property holds: for every cycle in G that contains e, the edge with the highest weight is not e.

   Direction 1: If e belongs in some MST of G, then for every cycle of G containing e, the edge wigh the highest weight is not e.

   Let M be any MST of G containing e. Let c be any edge in G but not M such that adding c to M produces a cycle containing e. By definition of cycle, the vertices connected by e now have a path to each other not using e. So, remove e from M. Because e is no longer in the cycle, the cycle is broken. This leaves a new spanning tree in the graph. This new spanning tree is identical to the previous spanning tree except it contains c instead of e. Because edge weights are distinct, this new spanning

tree with c is either lighter or heavier than the tree using e. If it were lighter, then that would contradict the assumption that M is an MST using e because it would have used c instead. So, it must be heavier. This implies that c is heavier than e and thus e is not the heaviest edge. This proves the assertion that for every cycle of G containing e, the edge with the highest weight is not e.

Direction 2: If for every cycle in G containing e the edge with the highest weight is not e, then e belongs in some MST of G.

Let C be any cycle in G containing e. Let h be the heaviest edge in that cycle. By the claim's assumption, assert that e and h are distinct. Consider any partitioning of the vertices in C such that the only two edges in C crossing the cut are e and h. By the cut property, the least of these edges will be in an MST of G. Because e is lighter than h, the construction of such an MST will choose e over h to connect the two cuts, implying that e will be in some MST of G.

(b) **(5 points)** Given an edge e in G, give an algorithm that outputs YES if there exists an MST of G that contains e, and NO otherwise. Your algorithm must have runtime asymptotically faster than the algorithms given in class for finding MSTs. You must describe your algorithm, prove its correctness and state its run time. You need not prove the runtime. (You may use the result from Part (a) even if you did not prove it.)

Consider this algorithm. In the algorithm, let v_1 and v_2 denote the two vertices connected by e:

i. Remove e from G.
ii. Initialize a Visited set and a PathWeights set for use in the DFS.
iii. Run a modified DFS on the graph starting at v_1. In the DFS, track the maximum-weight edge traversed on the current path. If the DFS ever reaches v_2, add the maximum weight of that path to the PathWeights set.
iv. Let G' be the G without v_2. If DFS did not visit every vertex in G', return NO.
v. If the PathWeights set is empty, return YES.
vi. If e is heavier than any value in PathWeights, return NO. Else, return YES.

Correctness:
If the algorithm returns something, it is correct about the MST membership of e: First, if the DFS didn't visit every vertex in G', then the graph is not connected even with e added back. The vertex v_2 is excluded because it might be unreachable without e. If the graph is not connected, then a spanning tree cannot exist, and e can subsequently not be part of such a tree. Here, it's correct to return NO. Otherwise, the DFS will have collected the maximum weight edges on every path from v_1 to v_2. If no such path exists, then e is the only edge the connect v_1 and v_2. For the tree to span, it must include e. So, it's correct to return YES. Finally, if e is lighter than the heaviest edge on every path from v_1 to v_2, then e is also not the heaviest edge in every cycle containing e. By part a, it is correct to return YES. Otherwise, NO is returned, which is also correct due to the iff nature of part a.

If e is in an MST of G, then the algorithm returns YES: If e is in an MST of G, then such an MST exists. This implies all vertices in the graph are connected, and the DFS will visit the entire graph except possibly v_2. So, the algorithm will not return NO from

that case. If e is the only edge to v_2 from the rest of the graph, then the algorithm will correctly return YES because e is required to make the tree span the entire graph. Otherwise, according to part a, because e is in an MST of G, the maximum-weight edges collected in the PathWeights set must all be greater than e. The final check will observe this and return YES.

Runtime:

The visited set is already part of DFS. Tracking the maximum weight edge along a path and maintaining PathWeights takes $O(1)$ time. So, DFS still takes time $O(E+V)$ where E is the number of edges and V is the number of vertices. This also bounds the size of PathWeights. Thus, the runtime is that of the DFS, $O(V+E)$.

2. **Maximal independent set in an evolving graph (30 points):**

Given an undirected graph $G = (V,E)$, we say that a subset $S \subset V$ is an *independent set* if no two vertices in S are connected by an edge. We say that S is a *maximal independent set (MIS)* if S is an independent set and furthermore there is no strict superset T (i.e., a set T with $S \subsetneq T$) which is also an independent set. In this problem we will explore the running time of algorithms computing and maintaining maximal independent sets in graphs with n vertices, m edges and with maximum degree Δ (i.e., every vertex $u \in V$ has at most Δ edges touching it.)

(a) **(5 points)** Give an algorithm that finds a maximal independent set of G, given G in the adjancency list representation. (You must describe your algorithm fully and give a brief explanation of why it is correct. You should state your runtime but you don't need to prove it.)

Consider this algorithm: Given an adjacency list graph called `adj`, initialize an empty set called `ind_set`. While `adj` is not empty, pop a vertex `v` and its edges from `adj` and add `v` to `ind_set`. Then, for every neighbor in edges, delete that neighbor from `adj`. At the end, return `ind_set`. Here's pseudocode for such an algorithm:

```
fn max_ind_set(adj: adjacency list graph) -> Set(vertices):
  let ind_set = {};
  while adj is not empty:
      let (v, edges) = pop entry from adj;
      ind_set.add(v);
      for neighbor in edges:
          adj.delete(neighbor);
  return ind_set;
```

Correctness:

Lemma, `ind_set` is an independent set and the vertices in `adj` share no edges with vertices in `ind_set`.

- Base case: `ind_set` is an independent set initially because it's empty. There are also trivially no edges between `ind_set` and `adj`.
- Inductive hypothesis: after $0 \leq n$ iterations of the `while` loop, (1) `ind_set` is an independent set and (2) vertices `adj` share no edges with vertices in `ind_set`.
- Inductive step: at the $n+1$ iteration of the while loop, `v` is popped from `adj`. By the inductive hypothesis, `v` shares no edges with vertices in `ind_set`. So,

adding it to `ind_set` maintains `ind_set` as an independent set. Then, every vertex sharing an undirected edge with `v` is deleted from `adj`. After this, `adj` contains no vertices with edges to `v`. By the inductive hypothesis, it also contains no edges with vertices previously in `ind_set`. This preserves the inductive hypothesis and thus proves the claim inductively.

Proof of correctness: From the lemma above, assert that once `adj` is empty, `ind_set` is an independent set and there are no vertices remaining in the graph that don't share an edge with a vertex in `ind_set`. This implies that the value returned from `max_ind_set` is a maximal independent set. Similarly, because a maximal independent set will always exist in a non-empty graph, if a maximal independent set exists in the input graph, one is found and returned by `max_ind_set`.

Runtime:
The algorithm iterates every vertex and every edge in the graph, so it runs in time $O(V+E)$. That assumes the input adjacency list is encoded as a HashMap with key vertex and value HashSet of neighbor vertices. I further assume insertion and deletion on those data structures take an average of $O(1)$ time.

Now suppose the graph G is changing over time, and we want to maintain a maximal independent set of this graph without having to recompute it from scratch every time G is updated. Concretely, suppose that in each time step, some edge is either added to or deleted from G. Our basic data structure simply maintains a set $S \subseteq V$ in the form of an array indexed by V such that $S[u] = 1$ if $u \in S$ and 0 otherwise.

(b) **(0 points, not to be turned in)** Prove that the adjacency lists can be maintained with $O(\Delta)$ cost per insertion and deletion. (You may assume you have a solution to this problem in future parts even if you did not solve it.)

(c) **(7 points)** Describe algorithms INSERT(e) and DELETE(e) to maintain S under edge insertions and deletions respectively. Give upper bounds on the runtime of both operations. (Your algorithm and its claimed run times must be correct, but you need not prove these. Note that for full credit your runtimes should depend only on Δ and not on n.)

Note: While it's not stated in the prompt, I'm assuming my methods have read access to the graph.

Consider these three functions:

- `add_if_able`: Accepts a vertex `v`. If no neighbors of `v` are in `S`, then add `v` to `S`.
- `insert`: Accepts an edge between vertices `v1` and `v2`. If both of the vertices are in `S`, then remove `v1` from `S` and call `add_if_able` on all of neighbors of `v1`.
- `delete`: Accepts an edge between vertices `v1` and `v2`. If exactly one of the edge endpoints is in `S`, call `add_if_able` on the vertex not in `S`.

Here's pseudocode for these algorithms:

```
DynamicMaximalIndSet {
    Independent set tracker S;

    fn add_if_able(v: Vertex, graph: &Graph):
        for neighbor in graph[v]:
            if S[neighbor] == 1:
                return
        S[v] = 1;

    fn insert((v1, v2): edge, graph: &Graph):
        if S[v1] != 1 or S[v2] != 1:
            return;
        S[v1] = 0;
        for neighbor in graph[v1]:
            add_if_able(neighbor, graph);

    fn delete((v1, v2): edge, graph: &Graph):
        if S[v1] == 0 and S[v2] == 0:
            return;
        Assert S[v1] and S[v2] are not both 1;
        let v = the side of the edge not in S;
        add_if_able(v, graph);
}
```

While proofs are not required, here's my reasoning for each along with runtime:

- `add_if_able`: If a vertex has no edges to vertices in the independent set, then it can be added to the independent set. This iterates the edges of a vertex. If each vertex has $O(\Delta)$ edges, then this takes $O(\Delta)$ time.
- `insert`: If the new edge is between two vertices not in `S`, both of those vertices must already have some neighbor in S preventing them from joining S. This requires no updates. Similarly, if one of the vertices is in `S`, then the other edge has yet another reason it cannot be in `S`. That case should also be igored. If both vertices are already in `S`, then one of them must be removed to maintain the independent set property. I arbitrarily chose `v1`. Removing `v1` might enable any of its neighbors to now join `S`, so they must be considered. However, either they cannot join because of a different neighbor in `S`, or they can join, giving all their neighbors yet another neighbor in `S`. Because of that, only one degree of separation from the initially-removed vertex must be considered. Considering all the neighbors of all the neighbors of the removed vertex takes time $O(\Delta^2)$.
- `delete`: If an edge is deleted between two vertices not in `S`, then both vertices still must have some neighbor in S preventing them from joining S. This implies no change. Simiarly, it should be impossible to delete an edge between two vertices in S because of the independent set property. Finally, if only one of the vertices is in `S`, the vertex not in `S` might now be able to join `S`. Checking this gives the function a total runtime of $O(\Delta)$.

(d) **(3 points)** For every integer $\Delta > 0$ describe an example (i.e., a graph, an MIS, and an edge to

be inserted) such that the number of queries to S for inserting an edge asymptotically match your upper bound from Part 2c.

Let G be a complete graph with n vertices. By definition, every vertex has $n-1$ edges, so Δ is $n-1$. Because it's a complete graph, only one vertex v_0 will initially be in S. Choose another vertex v_k in the graph and remove its edge with v_0. Because v_0 was the only vertex in S, v_k will now also be added to S. Now, re-insert that edge. Doing so will cause v_0 to be removed from the graph. Then, for each of v_0's neighbors v_{nb}, each of the v_{nb}'s neighbors will be inspected to see if v_{nb} can be added to the graph. Because the graph is complete, this will result in $(n-1)*(n-1)$ loop iterations, which will have $\Theta(\Delta^2)$ runtime. Allowing any $n > 1$ will demonstrate this behavior for any integer $\Delta > 0$.

To speed up the runtimes from Part 2c, suppose we decide to additionally maintain an array A indexed by V, such that for every $u \in V$, $A[u]$ counts the number of neighbors of u that are in S. (So $A[u] = |\{v \in V | v \in S, (u,v) \in E\}|$.)

(e) **(5 points)** Give algorithms A-INSERT and A-DELETE that maintains both S and A under edge insertions and deletions. (While any correct polynomial-time algorithm will get you full points, needlessly inefficient algorithms will lose points in the next part!)

Consider this set of algorithms:

- `add_if_able`: Given a vertex `v`, if `A[v]` is greater than zero, return. Else, add `v` to `S` and, for each `neighbor` of `v`, increment `A[neighbor]` by one.
- `insert`: Given an edge between `v1` and `v2`, if neither vertex is in `S`, return. If exactly one of the vertices `v` is in `S`, then increment `A[v]` by one and return. Else, remove `v1` from `S`, set `A[v1]` to one, and, for each neighbor of `v1` except `v2`, decrement `A[neighbor]` by one and call `add_if_able` on that `neighbor`.
- `delete`: Given an edge between `v1` and `v2`, if neither edge is in `S`, return. Assert that the vertices are not both in `S`. Let `v` be the side of the deleted edge not in `S`. Decrement `A[v]` by one and call `add_if_able` on `v`.

Pseudocode for this algorithm:

```
ADynamicMaximalIndSet {
    Independent set tracker S;
    Neighbor count tracker A;

    fn add_if_able(v: Vertex, graph: &Graph):
        if A[v] > 0:
            return;
        S[v] = 1;
        for neighbor in graph[v]:
            A[neighbor] += 1;

    fn insert((v1, v2): edge, graph: &Graph):
        if S[v1] == 0 and S[v2] == 0:
            return;
        if S[v1] == 0 or S[v2] == 0:
            let v = the side of the edge not in S;
            A[v] += 1;
            return;

        S[v1] = 0;
        A[v1] = 1;
        for neighbor in graph[v1] excluding v2:
            A[neighbor] -= 1;
            add_if_able(neighbor, graph);

    fn delete((v1, v2): edge, graph: &Graph):
        if S[v1] == 0 and S[v2] == 0:
            return;
        Assert S[v1] and S[v2] are not both 1;
        let v = the side of the edge not in S;
        A[v] -= 1;
        add_if_able(v, graph);
}
```

Justification for each method:

- `add_if_able`: If `v` has any neighbors in `S`, then it does not belong in `S` by definition of independent set. Otherwise, it should be added. After it's added, all of its neighbors' `A` counts must be updated to reflect that `v` is now in `S`.
- `insert`: If neither `v1` nor `v2` is in `S`, then no trackers need to be updated. If exactly one vertex in the edge is in `S`, then the side not in `S` should be updated to reflect its additional edge with a vertex in `S`. Otherwise, if both vertices are in `S`, then arbitrarily choose `v1` to be dropped. Doing so implies that `v1` now has one edge into `S` with `v2` and no others because it was previously in the independent set. Each non-`v2` neighbor of `v1` just lost an edge into `S` and should be decremented and reconsidered for membership in `S`.

- `delete`: This is almost identical to before except the vertex not in `S` now has its count into `S` decremented due to the edge deletion.

(f) **(10 points)** Assume that initially the graph G is empty (no edges), and S consists of all vertices in V. Give an amortized analysis proof that after T updates to G, the total runtime of the operations is at most $O(\Delta \cdot T)$. (Hint: Consider a charging scheme that charges the runtime of adding a vertex to S to the vertex itself. You should be careful to pay this charge when the same vertex is deleted from S!)

Let there be two types of events:

- Type 1: A vertex is added to S.
- Type 2: A vertex is removed from S.

Consider an amortization scheme where the cost of a Type 1 action is completely paid by a Type 2 action on the same vertex. Analyze each method in the data structure for its complexity:

- `add_if_able`: The first case is $O(1)$. Otherwise, in the worst case, this function runs in $O(\Delta)$ to update every neighbor of a vertex added to S. Because this only happens during a Type 1 event, defer all of this $O(\Delta)$ cost to the Type 2 action removing the same vertex from S. Doing so allows `add_if_able` to run in charged $O(1)$ time.
- `insert`: The first two cases are $O(1)$. Otherwise, if both vertices are in `S`, then `v1` must be removed from `S`. The Type 2 update itself is $O(1)$, but it also incurs the deferred Type 1 cost of $O(\Delta)$, which is now paid off. Then, for every neighbor of `v1`, perform an $O(1)$ update to `A` and call `add_if_able`. Under the charging scheme, `add_if_able` costs a charged $O(1)$ time, and calling that on every neighbor costs $O(\Delta)$. Thus, `insert` costs at most $O(\Delta)$.
- `delete`: The first case is $O(1)$. Otherwise, an $O(1)$ update is made followed by `add_if_able`. By the charging argument, `add_if_able` costs $O(1)$ now, so `delete`'s charged cost is $O(1)$.

The graph starts out with every vertex in `S`, so initial operations can only cause constant-time cases or vertices to be deleted from S. That implies paying for insertions that were never charged, which balances out later insertions that are never paid for through deletion. Otherwise, for every insertion that's eventually deleted, there's a charged cost of $O(\Delta)$ and a paid cost of $O(\Delta)$. Because the number of possible insertions and deletions bound each other, this yields a final cost across T operations of $O(T \cdot \Delta)$ time.

3. **Sorta sorting with heaps (24 points)** Explain how to solve the following two problems using heaps. (No credit if you're not using heaps!)

(a) **(12 points)** Give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list.

Note: My solution assumes (1) all of the k input lists are non-empty and (2) the lists are linked lists with $O(1)$ cost to push and pop at the head and tail.

Consider this algorithm: Given a list of linked lists, first initialize pointers for the output head and tail. Create a sentinel node at the head with value $-\infty$ and point the

tail to it. Also initialize a min heap for linked list nodes keyed by node value. Then, for each input linked list, push the head onto the heap. (Note: this could also use `heapify`, but it doesn't change asymptotic runtime). Finally, while the heap is not empty, pop a node, push its child back onto the heap if a child exists, set the popped node's child to empty, set the tail's next node to the popped node, and progress the tail pointer to the popped node that was just added. At the end, return the node after the sentinel. Pseudocode for this algorithm:

```
fn merge_sorted_lists(lists: List of Lists) -> List:
    let head = sentinel node(val = -inf, child = none);
    let tail = head;

    let heap = MinHeap of list nodes sorted by head node value;
    for list_head in lists:
        heap.push(list_head);

    while heap is not empty:
        let node = heap.pop();
        let child = node.next;
        if child is not empty:
            heap.push(child);
        node.next = empty list;
        tail.next = node;
        tail = tail.next;

    return head.next;
```

Correctness:
Begin by establishing the following claim: After $0 \leq m < n$ iterations of the while loop, (1) every node in the result list is less than or equal to every node in any list in the heap, (2) every node in the result list beginning at `head` is sorted, and (3) the length of the result list is $m+1$. Prove by induction:

- Base case: At iteration $m = 0$, `head` has not been modified. The sentinel's value is $-\infty$, which is less than every node in every list in the heap. A list of length 1 is trivially sorted, and the list has length $m+1 = 0+1 = 1$. This establishes the base case.
- Inductive hypothesis: Assume after $0 \leq m < n$ iterations of the while loop, the claim holds.
- Inductive step: At the $m+1$ iteration of the while loop, the `next` entry is popped from the heap, made childless, and added to the result list. If it exists, the child `next` is added back to the heap such that only one total node is removed from the combined lists of the heap. Each input list is sorted, so the k values used as keys in the heap are the least values of their lists. Because it's a min heap, the least of all these values is at the top of the heap, implying that popping it removes the least value among all nodes of all lists in the min heap. When this is popped and added to the result list, it is now less than or equal to every node

remaining in any list in the heap. This proves point (1). Because the popped node was in the heap at time m, assert by the inductive hypothesis that it was greater than every node in the result list. Consequently, because the result list was sorted least to greatest, when the popped node is added to the end of the result list, the result list remains in sorted least to greatest order with this new greatest node at the end. This proves point (2). By the inductive hypothesis, the size of the result list at step m was $m+1$. Adding this new node yields a result list of size $(m+1)+1$, which proves point (3). The inductive step is thus established, and, by induction, the claim must be true.

Applying the proof above, after the while loop has executed n times, the result list will be sorted and of length $n+1$. Removing the $-\infty$ sentinel node yields the n input values still sorted least to greatest. So, if a sorted order of n input values exists, the algorithm returns it.

The only thing this algorithm returns is the linked list beginning after head. by the induction proof above, such a list is a length-n list composed of the input elements in sorted order. So, the only thing the algorithm returns is a length-n sorting of the input elements.

By these two cases, the algorithm must be correct.
Runtime:
Initializing the linked list takes $O(1)$ time. Assuming the input lists are not empty, the time to iterate all input lists is bounded by the number of nodes in all lists, n. Each list requires one insertion into the heap, and the heap only ever has one entry for each of the k lists. So, each insertion into the heap requires time $O(\log k)$. This yields an initialization complexity of $O(n \log k)$. Using heapify under the same assumptions would take $O(n)$, but it doesn't reduce overall runtime.

Then, because one node is removed from one of the non-empty lists at each iteration of the `while` loop, the `while` loop iterates at most n times. For each iteration, a heap `pop` and `push` occur. Because the heap contains at most one entry for each of the k input lists, the heap never exceeds size k. This gives `push` and `pop` a cost of $O(\log k)$ time, resulting in a `while` loop complexity of $O(n \log k)$.

Thus, the total complexity of this algorithm is $O(n \log k)$.

(b) **(12 points)** Say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.

Consider this algorithm: given a list `nums` of k-close sorted numbers and an integer k, initialize a min heap and a `result` list. Fill the heap with the first k numbers in the input list (or the entire input list if k exceeds n). Then, while the heap is not empty, if there's a next number from the input, add it to the heap. After that, pop the min value from the heap and push the popped value to `result`. At the end, return the `result` array. Pseudocode for this algorithm:

```
fn k_close_sorted(nums: List(int), k: int) -> List(number):
    let heap = MinHeap(number);
    let index = 0;
    while index < k and index < nums.len():
        heap.push(nums[index]);
        index += 1;

    let result = [];
    while heap is not empty:
        if index < nums.len():
            heap.push(nums[index]);
            index += 1;
        result.push(heap.pop());
    return result;
```

Correctness:
Begin by establishing the following claim inductively: In the second while loop, when `result.len() == a` for some $0 \leq a \leq n$, (1) the entry belonging at position a in the array is initially in the heap and (2) the elements in the `result` array are sorted.

- Base case: Before the first push to the result array, the initial `while` loop added the first k elements of `nums` to the heap or the entire array, whichever is less. The `if` condition also adds another element if able. Thus, the first $k+1$ elements of the k-close sorted input list are in the heap. The definition of k-close requires that one of these values must be the first element in the correct sorted order, which validates point (1). The least of these is popped from the heap and added to the `result` array. Because it is one element, the `result` array is trivially sorted.
- Inductive hypothesis: The claim holds for all $0 \leq a < n$.
- Inductive step: When there are some a elements already in the `result` array, there are k elements in the heap within k of the $a+1$ element. The `if` condition adds the next element to the heap as well, meaning that the heap now contains the $k+1$ closest elements to index $a+1$ or every remaining unsorted element from the input list, whichever is less. This validates point (1) because the entry belonging at index $a+1$ now must be in the heap. The minimum among these is chosen from the heap and put at a location $a+1$ in `result`. If this value were in the heap at iteration a, then it must be greater than or equal to the value at a because it wasn't popped from the heap. If it were just added, it must be greater than a because it was more than k spaces from a. In either case, this newly pushed value must be greater than or equal to a. By the inductive hypothesis, the result array was already sorted least to greatest. Because this new value is in correct sorted order with the value at a, the list is still in sorted order. This proves the inductive step and validates the claim by induction.

Preceding the second `while` loop, the first `while` loop ensures the heap carries the first k elements of the array or the entire array, making the inductive proof above applicable.

Thus, once $a = n$, the heap will be empty and all n elements in the result array will be in sorted order. This implies that if a correct sorting exists from the input k-sorted list, it is returned.

Because the second `while` loop always chooses the minimum value from the heap and the heap is guaranteed to contain the value belonging at index a, the result array constructed by the algorithm will always produce a correct sorting. So, when the result array is returned from the algorithm, it must be a correct sorting.

Runtime:
Initializing the heap and `result` list takes constant time. Because the heap only ever contains at most $k+1$ elements, pushing and popping from the heap takes $O(\log k)$ time. Pushing the first k elements would normally require $O(k \log k)$ time. However, because the `while` loop exits early if k exceeds n, this step more accurately takes $O(n \log k)$ steps.

The second `while` loop includes some constant time work and two $O(\log k)$ operations on the heap. Because this loop executes once for each of the n elements belonging in the `result` list, this loop takes a total of $O(n \log k)$ time. In total, the algorithm runs in $O(n \log k)$ time.

4. **d-heaps (0 points, optional)**[1] Consider the following generalization of binary heaps, called d-heaps: instead of each vertex having up to two children, each vertex has up to d children, for some integer $d \geq 2$. What's the running time of each of the following operations, in terms of d and the size n of the heap?

   (a) delete-max()

   (b) insert(x, value)

   (c) promote(x, newvalue)

   The last operation, promote(x, newvalue), updates the value of x to $newvalue$, which is guaranteed to be greater than x's old value. (Alternately, if it's less, the operation has no effect.)

5. **Suboptimality of greedy algorithm for set cover (10 points)** Give a family of set cover problems where the set to be covered has n elements, the minimum set cover is size $k = 3$, and the greedy algorithm returns a cover of size $\Omega(\log n)$. That is, you should give a description of a set cover problem that works for a set of values of n that grows to infinity – you might begin, for example, by saying, "Consider the set $X = \{1, 2, 3, \ldots, 2^b\}$ for any $b \geq 10$, and consider subsets of X of the form...", and finish by saying "We have shown that for the example above, the set cover returned by the greedy algorithm is of size $b = \Omega(\log n)$." (Your actual wording may differ substantially, of course, but this is the sort of thing we're looking for.) Explain briefly how to generalize your construction for other (constant) values of k. (You need not give a complete proof of your generalization, but explain the types of changes needed from the case of $k = 3$.)

   Consider two colors of sets. Red sets are an optimal $k = 3$ decision, and blue sets are what the greedy algorithm chooses. These colors are also used in the pictures on the next page.

[1] We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.

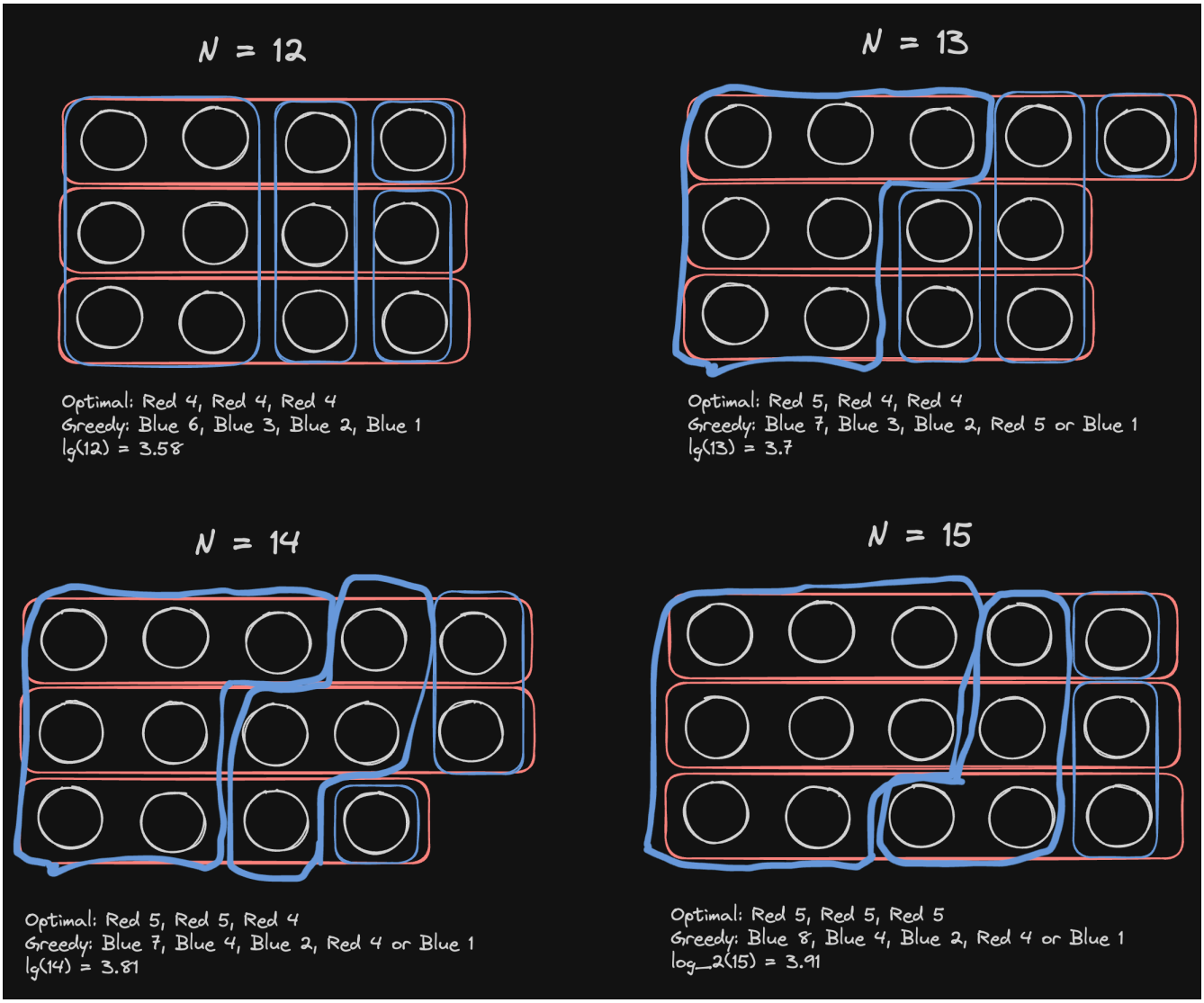
Given an input collection of n elements, create three red sets with $\frac{n}{3}$ elements partitioning the n input elements. Divide remainders such that the difference between the largest and smallest of these sets is at most 1.

Use this recursive procedure to construct the blue sets: Given some nonzero n elements that are not yet in a blue set, create a new blue set consisting of $ceil(\frac{n}{2})$ elements. Select these equally from the elements in red sets such that the difference between the most and least elements selected from a single red set is one. If all elements are now also in blue sets, exit. Otherwise, recursively continue this process on the $floor(\frac{n}{2})$ elements remaining outside of a blue set until every element is also in a blue set.

Assume this construction is applied to any large enough n, for example $n \geq 12$. For every grouping of this form, there is an optimal set cover of size 3 identified by the red sets. However, the first decision of the greedy algorithm never chooses a red set because the first blue set has $ceil(\frac{n}{2}) > ceil(\frac{n}{3})$ elements. Because blue elements are allocated equally from among the red sets, as n is recursively halved, this same decision is continually made until a set cover is produced of almost all blue sets. The last set may be either red or blue because it only needs to cover one element, but that doesn't affect the total set count.

The construction of blue sets halves n at each iteration, producing $\log n$ blue sets. Because the greedy algorithm chooses only blue sets until a single element remains, the greedy algorithm will always choose at least $\log n$ sets. Thus, $b = \Omega(\log n)$.

In the example graphs below, here's the decision process for $n = 12$: An optimal algorithm will choose the three red sets of size 4. However, the greedy algorithm will begin by choosing the blue set of size 6. After that, each red set would add 2 new elements. But there's a blue set covering 3 new elements, so that's added instead. Now, each red set adds only 1 new element. The blue set with 2 new elements is selected instead. Finally, the last element could be covered by either the single blue set or the top red set. Regardless, this leaves a greedy selection of size 4 when an optimal selection would only require 3 sets. (Examples on next page)



6. **Tracking components in an evolving graph (15 points)** You are secretly Gossip Girl, an anonymous gossip blogger who keeps track of friendships at Constance Billard High School. You publish an up-to-date map of the friendships at Constance on your website.[2] You maintain this map by a stream of distinct tips from anonymous followers of the form "A is now friends with B."

   (a) **(5 points)** You call some groups of people a "squad": each person is in the same squad as all their friends, and every member of a squad has some chain of friendships to every other member. For example, if Dan is friends with Serena, Serena is friends with Blair, and Alice is friends with Donald, then Dan, Serena, and Blair are a squad (You make up the name "The Gossip Girl Fan Club") and Alice and Donald are another squad ("The Constance Constants"). Give an algorithm that takes in a stream of (a) tips and (b) requests for a specified

[2] There are no ethical concerns here, because you're a character in a highly-rated teen drama.

person's squad name. You should answer requests that come in between tips consistently—if you make up the name "The Billard billiards players" for Dan's squad, and you're asked for Serena's squad's name before any new tips come in, you should report that it's "The Billard billiards players".

In this question, assume friendships are permanent. There's no mention of an unfriend event, it simplifies the algorithm, and it makes everyone's life better. Assume it's fine to name a squad by some member of the squad. This naming will consistently identify squads by the person at the root of their union find tree. So, if B's squad is merged with A's squad and A becomes the squad tree root, now B is in A's squad. The loss of B's squad name is inevitable anyway.

Consider a data structure holding a set of people, initially empty, and a union find data structure called `squads`. For friendship events, have a method `add_friendship` that takes two names, A and B. If A or B isn't yet in the set, add them. Then, call `make_set` to add them to `squads`. Finally, run `squads.union(A, B)`. For queries on a person A, call `squads.find(A)` and return the name of the root person as the name of the squad. Pseudocode:

```
SquadTracker {
    let squads = UnionFind data structure;
    let tracked_people = Set of names, initially empty;

    pub fn process_stream(event: Request) -> Response:
        return match event
            "A is now friends with B" => add_friendship(A, B),
            "request squad name for A" => get_squad(A)

    private fn add_friendship(A: Person, B: Person):
        for person in A and B:
            if person is not in tracked_people:
                tracked_people.add(person);
                squads.make_set(person);
        squads.union(A, B);

    private fn get_squad(A: person) -> SquadName:
        let person_at_root = squads.find(A);
        return "{person_at_root}'s squad";
}
```

Correctness:
First, prove by induction that every person in a squad is encoded in a single union find tree. (UnionFind holds a forest of such trees). Do so by induction on the number of calls made to `add_friendship`.

- Base case: Given people A_1 and B_1 for the first call to `add_friendship`, A_1 and B_1 are added to `tracked_people` and `squads`. Then, they're joined into a single squad. Because these are the only two people on the graph and they're correctly expressed as a squad, every squad is encoded as a single union find tree.

- Inductive hypothesis: After $0 \leq k \leq m$ calls to `add_friendship`, assume every person in a squad is encoded in a single UnionFind tree, where UnionFind holds a forest of such trees.
- Inductive step: At the $m+1$ call, if A_{m+1} and B_{m+1} are new people, the same logic as the base case applies and the rest of the graph will still be correct by the inductive hypothesis, making that case valid. If A_{m+1} or B_{m+1} is new and the other is already in the graph, only the new member will be created via `make_set` because the other will already be in `tracked_people`. If one person is already in the graph, they already have a friendship. So, the rank of the root of the squad tree of that user will be greater than 1, meaning that the new person will be added by union into the squad tree of the existing member. By the inductive hypothesis, every squad was correctly encoded before. Now, the squad of the already-present user just grew by one. Because that's the only squad the new user is a member of, this is a correct encoding of squads after the $m+1$ event. If both A and B already exist in the graph, both will already be a part of squad trees. Internally, the UnionFind structure will determine the greatest root and reparent the lesser squad into the greater squad. Because of the friendship between A and B, their squads will now be a single squad. Due to this union, all members of this new squad will identify by the shared root of their tree. Because this squad is now properly encoded as a single UnionFind tree and all other trees are untouched and already correct by the inductive hypothesis, this must leave a valid encoding after $m+1$ events. This proves the inductive step and the claim by induction.

After m calls to `add_friendship`, the proof above implies that squads now properly encodes the friend structure of the graph. So, if `add_friendship` is called, it makes a valid mutation to the friendship graph. And if `add_friendship` makes a mutation to the friendship graph, it only does so to encode a new piece of information. Because of this, `add_friendship` must be a correct handler for new friendship events.

Retrieving a squad name: Assert from the proof above that squads properly encodes the given friendships such that every member of a squad is in the same UnionFind tree. Thus, given a person A, `get_squad` will find A's squad tree and return the root of that tree as the squad name. Because every member of the union find tree will report the root of their tree as the squad name, every member of a squad will report being in the same squad. So, if person A is in the graph, A's squad will be correctly reported.

If any squad membership is reported from `get_squad`, it is the root of the tree parenting a squad that A is in. By the proof above, A must be in the same tree as every member of their squad. Thus, if anything is reported by `get_squad(A)`, it is the root of the tree that A is in, which is the name of A's squad. Because of this, `get_squad` must be correct.

Through these two handlers, the data structure correctly processes queries.

Runtime:
An `add_friendship` event requires up to two set `get` and `add` operations on a set, which are all constant time. There may also be two calls to `make_set` and one call

to `union`. A single call to `get_squad` makes one `find` request on the UnionFind structure. Thus, each operation performs a constant amount of work and makes a constant number of calls to UnionFind methods. So, if there are n people and m events in the stream, the total runtime guaranteed by the UnionFind data structure will be $O(m \log^* n)$.

(b) **(10 points)** A "circular squad" is defined to be a squad such that there is some pair of friends within the group that have both a friendship and a chain of friendships of length more than 1. In the example above, if Dan and Blair also became friends, then the group would be a circular squad. If Dan and Donald also became friends, they would all be in one circular squad. Modify your algorithm from the previous part so that you report names that contain the word "circle" for all circular squads (and not for any other squads).

7. **Greedy scheduling (35 points)** Consider the following scheduling problem: we have two machines, and a set of jobs $j_1, j_2, j_3, \ldots, j_n$ that we have to process. To process a job, we need to assign it to one of the two machines; each machine can only process one job at a time. Each job j_i has an associated positive integer running time r_i. The load on the machine is the sum of the running times of the jobs assigned to it. The goal is to minimize the completion time, which is the maximum of the load of the two machines.

   Suppose we adopt a greedy algorithm: for every i, job j_i is assigned to the machine with the minimum load after the first $i-1$ jobs. (Ties can be broken arbitrarily.)

   (a) **(5 points)** For all $n > 3$, give an instance of this problem for which the completion time of the assignment of the greedy algorithm is a factor of $3/2$ away from the best possible assignment of jobs.

   (b) **(15 points)** Prove that the greedy algorithm always yields a completion time within a factor of $3/2$ of the best possible placement of jobs. (Hint: Think of the best possible placement of jobs. Even for the best placement, the completion time is at least as big as the biggest job, and at least as big as half the sum of the jobs. You may want to use both of these facts.)

   (c) **(10 points)** Suppose now instead of 2 machines we have m machines and the completion time is defined as the maximum load over all the m machines. Prove the best upper bound you can on the ratio of the performance of the greedy solution to the optimal solution, as a function of m?

   (d) **(5 points)** Give a family of examples (that is, one for each m – if they are very similar, it will be easier to write down!) where the factor separating the optimal and the greedy solutions is as large as you can make it.