CS 124 Homework 3: Spring 2024

Anusha Murali

Collaborators: None

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

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.

   Solution: Assume that $e = (u, v)$ of a cycle in G belongs to some MST of G. Let us say that e has the highest weight of all the edges in that cycle. If we delete e from this MST, it will result in two connected subtrees, T_1 and T_2, where $u \in T_1$ and $v \in T_2$. Let us connect T_1 and T_2 using an edge $e' = (u', v')$ from the other path of the cycle, which does not include the highest weighted edge $e = (u, v)$. We choose $e' = (u', v')$ such that $u' \in T_1$ and $v' \in T_2$. Since e is the highest weighted edge in the cycle, the edge weight of e' is smaller than the edge weight of e.

Now, we have generated an MST, which has a lower weight than the MST that we started with. Since all MSTs of G have the same weight, this is a contradiction. So if an edge e belongs to some MST of G, then it is not the highest weighted edge in a cycle.

For the other direction, consider a cycle $C = (u_1, u_2, \ldots, u_n, u_1)$ in G, where one of the edges, (u_k, u_{k+1}), has the highest weight. Then we can construct a path including all the vertices in this cycle in two ways:

i. $u_k, u_1, u_2, \ldots, u_n, u_{k+1}$, which does not include the highest weighted edge, (u_k, u_{k+1}), or

ii. $u_1, u_2, \ldots, u_n, u_k, u_{k+1}$, which includes the highest weighted edge, (u_k, u_{k+1}).

Since path (i) has lower weight than path (ii), no MST will include path (ii). So the highest weighted edge of a cycle does not belong to any 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.)

Solution: We will use the results from Part (a): An edge e belongs to an MST of G, if and only if e is not the highest weighted edge in a cycle of G.

Let $e = (u, v)$. Let us run a modified depth first search (DFS), using u as the starting vertex. In this modified DFS, we will only traverse the edges that have smaller weights than $e = (u, v)$.

Let $u, u_1, u_2, \ldots u_k$ be the sequence of vertices traversed by the above DFS. If v is present in this sequence, then $e = (u, v)$ is not part of any MST of G, as there is a path from u to v, that consists of edges all of which have smaller weight than e. So $e = (u, v)$ is the highest weighted edge in a cycle of G. According to the results of Part (a), e cannot be part of any MST of G. So, we output NO.

If v is not present in the above sequence, then $e = (u, v)$ belongs to an MST of G as we couldn't find any smaller weighted edges than e to connect u and v. So, we output YES.

Correctness: In our DFS algorithm, we start from u and only traverse those edges that have smaller weight than e. If we found a cycle, then we have found a back edge (so we output NO) or terminate without finding one (so we output YES). This is analogous to running DFS on a graph, G', which is obtained by deleting all the edges smaller than the weight of e from G. Therefore, correctness follows from the correctness of DFS.

Runtime: The runtime of our DFS algorithm is $O(|V| + |E|)$, because we examine each vertex once, and each out-edge that has smaller weight than e from a given vertex once. We assume the comparison of the weight of an edge to that of e takes $O(1)$.

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.)

   Our algorithm is the following:

Algorithm 1 FIND-MAXIMAL-INDEPENDENT-SET(G)

```
Let S ← ∅ and T ← V
while T ≠ ∅ do
    Pick a vertex v ∈ T and add it to S
    For every (v, u) in v's list, remove u from T
return S
```

   Regarding the accuracy, inductively, whenever we remove a vertex from T, we remove all its neighbors from T, as well. So, S has to be an independent set. Now, because all the vertices removed are adjacent to some vertex in S, there cannot be a strict superset of S that is also an independent set. Therefore, S is an *MIS*.

   In terms of the runtime, depending on the data structure provided to us for each adjacency list, our runtime would be determined. If V is like a hash-map, then the runtime would be $O(m+n)$ because we are iterating over n vertices and m edges only once. The removal time would be $O(1)$ and checking whether a vertex is in T or not would be $O(1)$, as well.

   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.)

The INSERT(e) operation, where $e = (u, v)$, can either not change the MIS at all, or can decrease the size of S. When we add the edge e, we check if u and v are in S or not. If both are not in S, then we don't do anything because adding this edge is not affecting the adjacency of any of the vertices in S. If only one of them (WLOG u) is in S, then we still don't do anything because v was already eliminated because it was adjacent to some $u' \in S$. If both u and v are in S, then eliminate (WLOG) v. We still have an independent set at this point, but not necessarily an MIS. So, at this point, we work with all the neighbors of v. For each neighbor v' of v, if v' has no neighbor in $S = S \setminus \{v\}$, then we add v' to S (so, $S = S \cup \{v'\}$). We do this in sequentially. At the end of all the iterations, any other vertex outside S must have a neighbor in S, and no vertices in S are adjacent to one another. This gives as an updated MIS. The running time of this is $O(\Delta^2)$ because we are checking for at most Δ neighbors of v and each of that neighbor may have at most Δ neighbors. All look-up's are in $O(1)$, so this gives the said time complexity.

The deletion (DELETE(e)) algorithm has similar ideas as the above, but with a few small differences. If the edge $e = (u, v)$ is such that neither u nor v are in S, then we don't have to do anything because this doesn't affect the adjacency with any member in S. If exactly one of them (WLOG, u) is in S, then check if v has any neighbors that are in S. If not, then we add v to S. Otherwise, we still have an MIS. If S is already an independent set, then it cannot be the case that u and v are both in S. The running time of this algorithm is $O(\Delta)$ because in the worst case, we simply make Δ queries to S, which are all $O(1)$ time.

(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 the graph G have two components (G_1, G_2), which are alike. Each component has a root vertex r_1, r_2. From r_1, $\Delta - 1$ edges emerge that connect to different vertices $c_1^1, \ldots, c_{\Delta-1}^1$. From each c_i^1, another $\Delta - 1$ edges emerge into new vertices $d_{i,1}^1, \ldots, d_{i,\Delta-1}^1$. Each $d_{i,j}^1$ is connected to r_1. The same construction is there for r_2, as well. Now, let the MIS be $\{r_1, r_2\}$ (this holds because r_1 is adjacent to all the edges in its component, and r_2 is adjacent to all the edges in its own component). We claim that when we add an edge (r_1, r_2), we match the upper bound in the above algorithm asymptotically, and this holds for all $\Delta \in \mathbb{N}$. The algorithm would take one of the two roots (say r_2) out of the MIS S, and then look at all its neighbors c_i^2, which are $\Delta - 1$ in number, and for each i, it would check if the $d_{i,j}^2$ are in S or not. This would result in $(\Delta - 1)^2$ queries to S, which shows our claim.

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!)

For A-INSERT(e), we have an additional data structure to handle, where $e = (u, v)$. This is, in principle, the same as INSERT(e), but with a few subtle differences. When checking whether a vertex v' is in S or not, we need to check if $A[v']$ is equal to 0 or not, respectively. The first

case is when exactly one of u, v (say u) is in S. In that case, we saw that S will not change, but we would just increment $A[v]$ by 1. In the case where u, v are in S, and we are removing v from S, we have a step in INSERT(e), where we check the neighbors of neighbors of v that are in S. For that, we can use A. Only when we add a vertex v' in S, we simply make an increment to all the neighbors of v' in A, and when we remove a vertex v' from S, we decrease the values of all the neighbors of v' that are outside S by 1 in A, and increase the value of v' in A by 1. These aforementioned additions and deletions from S have been described in Part C.

In the deletion algorithm (A-DELETE(e)), when we remove an edge $e = (u, v)$, we only have to think about the case where exactly one of u and v (say u) is in S. We then decrement $A[v]$ by 1, and if $A[v]$ is 0, then we put v in S. In this case, we increment the A value of each neighbor of v by 1.

(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!)

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.

   Solution: Assume that each of the k lists is sorted in ascending order, so that the first element of each list contains the smallest in that list. Build a Min-Heap of k elements, where the k elements are the given k lists. We use the first element in each list as the key value to build the Min-Heap. Building a Min-Heap of k element takes $O(k)$ time.

   We remove the first element of the list at the top of the Min-Heap using the HEAP-EXTRACT-MIN operation, and add it to an initially empty output list. Removal of the first element of the list found at the top of the Min-Heap takes $O(1)$ time. Since the first value of the list at the top of the Min Heap has now changed, HEAP-EXTRACT-MIN operation will Min-Heapify the heap, which takes $O(\log k)$ time as it involves at most k items. (This is analogous to the MAX-HEAPIFY algorithm from CLRS).

   We will again remove the first element of the list at the top of the Min-Heap using the HEAP-EXTRACT-MIN operation and add it to the growing output list. We repeat the above steps n times until we have added all n elements to the output list. Therefore, the total runtime of merging k sorted lists is $O(k) + O(n \log k) = O(n \log k)$.

Algorithm 2 MERGE-K-SORTED-LISTS($A_1, A_2, \ldots, A_k$)

```
1: OUTPUT = []                                          {Initialize the output list}
2: X = BUILD-MIN-HEAP(A_1, A_2, ..., A_k)               {Use 1st element of A_i as the value to build Min-Heap}
3: for i = 1 to n do
4:     OUTPUT.append(HEAP-EXTRACT-MIN(X) )              {Extract the value from the root of Min-Heap}
5: return OUTPUT
```

Correctness: We shall use strong induction to prove that the OUTPUT list contains all n numbers in sorted order.

Base case: The base case is an empty OUTPUT list and the OUTPUT list of one element, both of which are trivially sorted.

Inductive hypothesis: We assume that the OUTPUT list of size m, for $i = 1, \ldots, m$. is in ascending sorted order.

Inductive step: Consider $i = m + 1$. Since HEAP-EXTRACT-MIN() always returns the minimum element from the heap, the m-th element was the smallest element when it was extracted, and is smaller than the $m+1$-st element, which is yet to be extracted from the Min-Heap. When we add $m+1$-st element to the OUTPUT list, it is therefore larger than m-th element, and is the smallest element among all the elements in the Min-Heap. Therefore, after the addition of the $m+1$-st element to the OUTPUT list, the OUTPUT list from $i = 1 \ldots m+1$ are in ascending sorted order. Since the inductive hypothesis is true for the base cases, and is true for $i = 1, \ldots m, m+1$, it follows that the OUTPUT list for $i = 1, \ldots n$ is in ascending sorted order at the end of the algorithm. □

Runtime: The BUILD-MIN-HEAP operation on line 2 (which is analogous to the BUILD-MAX-HEAP algorithm in CLRS) takes $O(k)$ time as it builds a Min-Heap of size k. The HEAP-

EXTRACT-MIN(X) call on line 4 takes $O(\log k)$ time as it involves extracting the root of the Min-Heap in $O(1)$ time and Min-Heapifying the changed heap in $O(\log k)$ time. Since HEAP-EXTRACT-MIN(X) is called n times, the total cost of running the for loop is $O(n \log k)$. Therefore the total runtime of merging k sorted lists is $O(k) + O(n \log k) = O(n \log k)$. □

(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.

Solution: In a k-close to sorted list, each number in the list is less than k positions left or right from its actual place in the sorted order.

The additional information that each number is found to be less than certain number of positions of its actual place allows us to construct an algorithm that is faster than the traditional $n \log n$ algorithm for sorting using heaps. Following steps outline our $O(n \log k)$ algorithm.

i. Build a Min-Heap using the first k numbers. This step takes $O(k)$ using the BUILD-MIN-HEAP function (which is analogous to the BUILD-MAX-HEAP() function in CLRS).
ii. Remove the number at the root of the Min-Heap (using HEAP-EXTRACT-MIN(), which also involves MIN-HEAPIFY()), takes $O(1) + O(\log k)$ time and add as the first element of the output array.
iii. Pick the next number in the input list and insert into the Min-Heap (using MIN-HEAP-INSERT(), which takes $O(\log k)$ time). This step bubbles up the smallest number to the root of the Min-Heap
iv. Repeat the above two steps until the input array becomes empty.

Therefore, the total runtime for our algorithm is $O(k) + (n-k)O(\log k) = O(n \log k) + O(k) - O(k \log k)$.

Since $n > k$, the runtime can be expressed as $O(n \log k) + O(k) - O(k \log k) = O(n \log k)$.

Algorithm 3 SORT-K-CLOSE-LIST(A)

```
1: OUTPUT = []                                          {Initialize the output list}
2: X = BUILD-MIN-HEAP(A[1], A[2], ..., A[k])            {Use the first k elements of A to build a Min-Heap}
3: for i = 1 to n do
4:     OUTPUT.append(HEAP-EXTRACT-MIN(X) )              {Extract the value from the root of Min-Heap}
5:     if (i ≤ n − k) then
6:         MIN-HEAP-INSERT(A[k + i])
7: return OUTPUT
```

Correctness: Let us consider the following loop invariant: At the start of the i-th iteration of the for loop on line number 3, the root of the Min-Heap X contains i-th smallest number of the input array A, and each of the other $k-1$ numbers in the Min-Heap are less than k positions above the root of the Min-Heap, and the OUTPUT array contains the first i smallest elements of the input array A.

Initialization: Prior to the first iteration of the for loop on line number 3, OUTPUT array is empty and X contains the first k elements of the k-close list, which is a Min-Heap created by calling BUILD-MIN-HEAP, with the smallest element at the root of the heap.

Maintenance: Because $X[1..k]$ is a Min-Heap, the smallest element out of the k elements is in $X[1]$. We also note that since the original input array A represents a k-close list, the smallest element in the entire input array $A[1,2,\ldots,n]$, is found within the first k positions. Hence BUILD-MIN-HEAP will move this smallest element to the root of the Min-Heap. During each iteration, the root of the Min-Heap, which represents the smallest number in the heap, is extracted using HEAP-EXTRACT-MIN() and appended to the OUTPUT array in that order, thereby maintaining the ascending order of the elements in the OUTPUT array. For $i < n-k$, we also insert the $A[k+i]$ from the input array into the Min-Heap using MIN-HEAP-INSERT(), which min-heapifies the entire Min-Heap once again so that the smallest element in the heap bubbles up to the root.

Termination: At termination, $i = 1$. The Min-Heap is empty as we added all the n elements from the input array A into the Min Heap, extracted them from the heap and added all n elements into the OUTPUT array in the order in which they were extracted. Therefore, the OUTPUT$[1:n]$ array is sorted in ascending order.

Runtime: BUILD-MIN-HEAP on line number 2 takes $O(k)$ time as it involves k elements. Both HEAP-EXTRACT-MIN() on line number 4 and MIN-HEAP-INSERT() on line number 6 take $O(\log k)$ time respectively. HEAP-EXTRACT-MIN() is executed n times and MIN-HEAP-INSERT() is executed $n-k$ times. Therefore, noting $n > k$, the total runtime is $O(n\log k) + O(k) - O(k\log k) = O(n\log k)$. □

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.)

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

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 $k = 3$.)

Solution WLOG, let us consider the universe of $n = 3(2^b - 1)$ elements, where $b \geq 4$. The universe is represented by the set of integers $X = \{1, 2, 3, \ldots, 3(2^b - 1)\}$ for any $b \geq 4$.

Consider the subsets of X of the form

$$\begin{aligned} S_1 &= \{1, 2, 3\}, \\ S_2 &= \{4, 5, \ldots, 9\}, \\ S_3 &= \{10, 11, \ldots, 21\}, \\ &\vdots \\ S_b &= \{3(2^{b-1} - 1) + 1, \ldots, 3(2^b - 1)\}, \end{aligned}$$

where the first subset, S_1 consists of the $3 \cdot 2^0 = 3$ integers starting from 1, and the subset S_k consists of the next $3 \cdot 2^{k-1}$ integers, with the last subset S_b containing the last $3 \cdot 2^{b-1}$ integers.

In addition, there are also the following 3 subsets present:

$$\begin{aligned} T_1 &= \{1, 4, 7, \ldots, 3 \cdot 2^b - 5\}, \quad \text{where } |T_1| = 2^b - 1 \\ T_2 &= \{2, 5, 8, \ldots, 3 \cdot 2^b - 4\}, \quad \text{where } |T_2| = 2^b - 1 \\ T_3 &= \{3, 6, 9, \ldots, 3 \cdot 2^b - 3\}, \quad \text{where } |T_3| = 2^b - 1 \end{aligned}$$

Let $\mathscr{F} = \{S_1, S_2, \ldots, S_b, T_1, T_2, T_3\}$. The greedy algorithm picks, at each stage, the set S that covers the greatest number of remaining elements that are uncovered.

Algorithm 4 Greedy-Set-Cover($X, \mathscr{F}$)

```
1: U = X
2: C = ∅
3: while U ≠ ∅ do
4:     select an S ∈ F (S is either S_i or T_i) that maximizes |S ∩ U|
5:     U = U − S
6:     C = C ∪ {S}
7: return C
```

Initially $U = X$. When the greedy algorithm runs, since $|S_b| = 3 \cdot 2^{b-1} > 2^b - 1$, the subset S_b will be selected over subsets T_1, T_2 or T_3. So $U = U - S_b$, and now there are $3(2^b - 1) - 3 \cdot 2^{b-1} = 3(2^{b-1} - 1)$ integers left in U. So, after the 1st iteration of the while loop, the cover is $\mathscr{C} = \{S_b\}$, and $U = U - S_b$.

The next largest subset that has size less than $|U|$ is S_{b-1} and the cover becomes $\mathscr{C} = \{S_b, S_{b-1}\}$, and $U = U - S_{b-1}$. The greedy algorithm continues in this manner, adding $S_b, S_{b-1}, S_{b-2}, \ldots, S_2, S_1$ in this order to $\mathscr{C}$ until U becomes empty.

Hence the greedy algorithm adds b subsets to find the set cover for this problem. Since $n = 3(2^b - 1)$, we find $b = \log(n+3) - \log 3$. As the greedy algorithm takes a minimum of b subsets to find the cover, we have shown that for the example above, the set cover returned by the greedy algorithm is of size $b = \Omega(\log n)$.

On the other hand, we note that the optimal cover is provided by the subsets T_1, T_2 and T_3. So the minimum set cover size is $k = 3$.

Generalization of the construction: We can generalize the above problem to any arbitrary minimum set cover of size k by considering a universe of $n = k(2^b - 1)$ elements, $X = \{1, 2, 3, \ldots, k(2^b - 1)\}$, where the b subsets S_i will be of the following shapes:

$$\begin{aligned}
S_1 &= \{\text{First } k \text{ integers}\},\\
S_2 &= \{\text{Next } k \cdot 2^1 \text{ integers}\},\\
S_3 &= \{\text{Next } k \cdot 2^2 \text{ integers}\},\\
&\vdots\\
S_b &= \{\text{Next } k \cdot 2^{b-1} \text{ integers}\},
\end{aligned}$$

and the k subsets T_i will be of the following shapes:

$$\begin{aligned}
T_1 &= \{1, k+1, 2k+1, \ldots, k \cdot 2^b - 2k + 1\}, \quad \text{where } |T_1| = 2^b - 1\\
T_2 &= \{2, k+2, 2k+2, \ldots, k \cdot 2^b - 2k + 2\}, \quad \text{where } |T_2| = 2^b - 1\\
&\vdots\\
T_k &= \{k,\ 2k,\ 3k,\ \ldots,\ k \cdot 2^b - k\}, \qquad\qquad \text{where } |T_k| = 2^b - 1
\end{aligned}$$

Even though the optimal set cover is provided by $T_1 \cup T_2 \cup \cdots \cup T_k$, the greedy algorithm will pick $S_b,\ S_{b-1},\ \ldots,\ S_2,\ S_1$ in this order, as at line number #4 of the **Greedy-Set-Cover** algorithm, each time, S_i will provide the cover for the largest number of uncovered elements so far, resulting in a suboptimal set cover of $\Omega(\log n)$.

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 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".

   Solution: We can implement this problem using the disjoint-set (union-find) operations. We first list the pseudocode for the functions required for the disjoint-set operations. Please note that the union operation is done by the MAKE-FRIENDS-WITH() function, which puts x in the same group as y and assigns y's parent as the parent of x. We use the name of the person at the root as the name of the squad. Let us assume the name(x) is the name of person x.

Algorithm 5 MAKESET(x)

```
p(x) ← x
rank(x) ← 0
name(x) ← x
```

Algorithm 6 FIND-SQUAD(x)

```
if x ≠ p(x) then
    p(x) ← FIND-SQUAD(p(x))
return name(p(x))
```

Algorithm 7 LINK(x, y)

```
if rank(x) > rank(y) then
    return LINK(y, x)
else if rank(x) = rank(y) then
    rank(y) ← rank(y) + 1
p(x) ← y
return y
```

Algorithm 8 MAKE-FRIENDS-WITH(x, y)

```
LINK(FIND(x), FIND(y))
```

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

The following algorithm takes in a stream of tips input in the form of a graph, where the edges between two vertices indicate friendships between the two people corresponding to the vertices.

Algorithm 9 MAKE-SQUADS(G)

```
1: for each vertex v ∈ G.V do
2:     MakeSet(v)
3: for each edge (u, v) ∈ G.E do
4:     if FIND-SQUAD(u) ≠ FIND-SQUAD(v) then
5:         LINK(u, v)
```

As an example, let us say that Dan is friends with Serena, and Serena is friends with Blair. The following stream of 2 operations will be done to make them into one squad, which will be named "Dan".

(1) MAKE-FRIENDS-WITH(MAKESET('Dan'), MAKESET('Serena'))

(2) MAKE-FRIENDS-WITH(MAKESET('Serena'), MAKESET('Blair'))

Now, the call to FIND-SQUAD('Blair') will return 'Dan' as the squad's name.

(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).

Solution: The following function PRINT-CIRCULAR-SQUAD(G) prints "circle" (along with the names of the people in the circular squad) if a squad contains some pair of friends within the squad that have a friendship and a chain of friendships of length more than 1.

Algorithm 10 PRINT-CIRCULAR-SQUAD(G)

```
1: for i = 1 to |G.V| do
2:     parent[i] = i                          {Initialize the parent pointer to the vertex itself}
3:     alreadyReporteed[i] = False.           {This person was not yet reported in the output}
4: for each vertex v ∈ G.V do
5:     MakeSet(v)                             {Make a new set for v, with parent pointing to v}
6: for each edge (u, v) ∈ G.E do
7:     uSet = FIND(PARENT, u)                 {Find the representative of the set containing u}
8:     vSet = FIND(PARENT, v)                 {Find the representative of the set containing v}
9:     if uSet == vSet then
10:        if not alreadyReported(uSet) then
11:            PRINT("circle", name(uSet))    {A cycle was found for the first time for this squad, so report it}
12:            PRINT(name(u), name(v))        {Also print the names of the persons}
13:            alreadyReported(uSet) = True
14:    else
15:        UNION(parent, uSet, vSet)          {Place them in the same squad}
```

If there are no cycles in a squad, IS-CIRCULAR-SQUAD() just prints the names of the people in that squad. The above function, IS-CIRCULAR-SQUAD() uses the following FIND() and UNION() functions.

Algorithm 11 FIND(parent, v)

```
if parent[v] != v then
    return FIND(parent, parent[v])
return v
```

Algorithm 12 UNION(parent, u, v)

```
uSetParent = FIND(parent, u)        {Find the representative of the set containing u}
vSetParent = FIND(parent, v)        {Find the representative of the set containing v}
parent[vSetParent] = uSetParent     {Make parent of vSetParent to be uSetParent}
```

Correctness: We show the correctness of IS-CIRCULAR-SQUAD() as follows:

Loop invariant: If an edge (u, v) is part of a squad, after k iterations of the **for** loop on line number 6, then u and v are in the same squad.

Initialization: Initially, all the vertices are in their own disjoint sets, and they are their own parents.

Maintenance: Before the k-th iteration of the **for** loop on line number 6, the vertices u and v are in the same squad, if and only if they have the same parent. If they are in the same squad, during the k-th iteration of the **for** loop, if the FIND() operation on u and v returns the same parent for both calls (on line number 7 and 8), then the edge (u, v) completes a cycle in the same disjoint set containing the vertices u and v, and "circle" will be printed along with the squad's name (line number 11) and the name of the people representing that edge (line number 12). If they are not in the same squad, due to the fact they are connected by an edge, the UNION() operation on line number 15 will place them in the same squad.

Termination: The **for** loop on line number 6 terminates after iterating through all $|G.E|$ edges (friendships). At termination, all of the edges in G have been visited and the parents of their end vertices were compared (on line number 9) to look for any cycles. □

Runtime: The **for** loop on line number 6 iterates once for each edge in G. During each of its iteration, the FIND() operation is called twice on the source and destination vertices of the edge, and the UNION() operation is called once to place vertices from disjoint sets into one squad. Using the union by rank algorithm and path compression from Part (a) the PRINT-CIRCULAR-SQUAD() therefore takes a total of $O(m \log^* n)$ time. □

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.

   Let M_1 and M_2 be the two machines. If the time taken to process the n jobs on M_1 and M_2 by the greedy algorithm is T and the time taken for the optimal algorithm is OPT, we are looking for an instance of the problem which would result in $T = \left(\frac{3}{2}\right)$OPT.
   For $n = 4$, Consider the following run times for the job sequence: 2, 1, 1, 4. The assignment using the greedy algorithm, which assigns the jobs to the two machines in the given order is:

M_1	M_2
2	1
4	1

   Therefore, the greedy algorithm requires $T = 2 + 4 = 6$ units of time. On the other hand, the best possible assignment is as follows:

M_1	M_2
4	2
	1
	1

   Hence, the optimal assignment requires OPT = 4 units of time. So, we find $T = \left(\frac{3}{2}\right)$OPT.
   The **key observation** from this example is that if there are n jobs to be processed, the processing time for the first $n-1$ jobs (equally divided between the two machines) must be exactly equal and the processing time for the n-th job should be equal to the time spent for processing the first $n-1$ jobs. In order to divide the runtime equally between the two machines, for odd n, the first $n-1$ job will have runtimes, $1, 1, 1, \ldots, 1$ and the last job will have runtime $n-1$. For even n, the first $n-1$ runtimes will be, $2, 1, 1, \ldots, 1$ and the last runtime will n.
   Therefore, the construction of a possible sequence of runtimes for odd and even n jobs are as follows:

   - For odd n: The first $(n-1)$ runtimes are all 1's and the last runtime is $(n-1)$.

     $$1,\ 1, \ldots,\ 1,\ (n-1)$$

   - For even n: The first runtime is 2, the next $(n-2)$ runtimes are all 1's and the last runtime is n.

     $$2,\ 1,\ 1,\ \ldots,\ 1,\ n$$

The following table shows a few example sequences using the above instance of the problem:

n	Job Runtimes	Greedy Algorithm	Optimal Algorithm	T/OPT
4	2, 1, 1, 4	6	4	3/2
5	1, 1, 1, 1, 4	6	4	3/2
6	2, 1, 1, 1, 1, 6	9	6	3/2
7	1, 1, 1, 1, 1, 1, 6	9	6	3/2
8	2, 1, 1, 1, 1, 1, 1, 8	12	8	3/2
9	1, 1, 1, 1, 1, 1, 1, 1, 8	12	8	3/2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Even n	$2,1,\ldots,1,n$	$3n/2$	n	3/2
Odd n	$1,1,\ldots,1,(n-1)$	$3(n-1)/2$	$(n-1)$	3/2

We can show that the above sequences for the runtimes for odd and even n, $n > 3$ using induction. The basis cases are $n = 4$ and $n = 5$, both of which show that the greedy algorithm is (3/2) times longer than the optimal algorithm. Our inductive hypothesis is for an arbitrary $k, k > 5$, the greedy algorithm is (3/2) times longer than the optimal algorithm. For the inductive step, there are two cases: k is either even or odd. In both cases, the proof follows the same arguments. Let us consider $n = k+2$. Hence, as can be seen from the table above, two additional jobs with runtimes = 1 will need to be assigned to M_1 and M_2 and the load for the last job is $k+2$. Hence the time taken for the greedy algorithm will increase by $2/2+(k+2-k) = 3$ time units. On the other hand, the optimal algorithm will only increase by $k+2-k) = 2$ time. Hence the ratio of the time taken for the greedy and the optimal algorithms for $n = k+2$ remains 3/2. □

(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.)

Solution: We use the hint in the problem, which is "even for the optimal 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".

Let M_1 and M_2 be the two machines and we use the greedy algorithm: for every i, job j_i is assigned to the machine with the minimum load after the first $i-1$ jobs, breaking times arbitrarily.

We assign jobs to M_1 and M_2 such that job j_i is assigned to the machine with the minimum load just before the assignment. Let us say that the last job, j_n, was run on M_1 and let T be the time when the last job was finished.

From the greedy strategy, we know that when the last job, j_n, was assigned, M_1 had a smaller load than M_2. Therefore, using the hint, the load on M_1 at this instance was at most the average load on both machines. In other words,

$$T - r_n \leq \frac{r_1 + r_2 + \cdots + r_{n-1}}{2}.$$

Therefore, we find,

$$\begin{aligned} T &\leq \frac{r_1 + r_2 + \cdots + r_{n-1}}{2} + r_n \\ &= \frac{r_1 + r_2 + \cdots + r_{n-1} + r_n}{2} + r_n - \frac{r_n}{2} \end{aligned}$$

But, we know that,

$$\text{OPT} = \frac{r_1 + r_2 + \cdots + r_{n-1} + r_n}{2} \quad \text{and} \quad r_n \leq \text{OPT}$$

Therefore, the time T when the last job was finished is,

$$\begin{aligned} T &= \frac{r_1 + r_2 + \cdots + r_{n-1} + r_n}{2} + r_n - \frac{r_n}{2} \\ &= \text{OPT} + \frac{r_n}{2} \\ &\leq \text{OPT} + \frac{\text{OPT}}{2} = \left(\frac{3}{2}\right)\text{OPT}. \end{aligned}$$

Hence the greedy algorithm always yields a completion time within a factor of 3/2 of the best possible placement of jobs. □

(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?

Solution: Given that we now have m machines, we assign jobs to $M_1, M_2, \cdots, M_m$ such that job j_i is assigned to the machine with the minimum load just before the assignment. WLOG, let us say that the last job, j_n, was run on M_1 and let T be the time when the last job was finished. Therefore, we can make the following observation:

- $\text{OPT} \geq \frac{\sum_{i=1}^{n} r_i}{m}$. If this is not true, the optimal time to process the jobs will be strictly less than $\sum_{i=1}^{n} r_i$, which is a contradiction.

From the greedy strategy, we know that when the last job, j_n, was assigned, M_1 had a smaller load than the loads on the other $m-1$ machines. We break ties in favor of M_1 (lower indexed machine). Therefore, using the hint, the load on M_1 at this instance was at most the average load on all the machines. In other words,

$$T - r_n \leq \frac{r_1 + r_2 + \cdots + r_{n-1}}{m}.$$

Therefore, we find,

$$\begin{aligned} T &\leq \frac{r_1 + r_2 + \cdots + r_{n-1}}{m} + r_n \\ &= \frac{r_1 + r_2 + \cdots + r_{n-1} + r_n}{m} + r_n - \frac{r_n}{m} \end{aligned}$$

But, we know that,

$$\text{OPT} = \frac{r_1 + r_2 + \cdots + r_{n-1} + r_n}{m} \quad \text{and} \quad r_n \leq \text{OPT}$$

Therefore, the time T when the last job was finished is,

$$\begin{aligned} T &= \frac{r_1 + r_2 + \cdots + r_{n-1} + r_n}{m} + r_n - \frac{r_n}{m} \\ &= \text{OPT} + r_n\left(1 - \frac{1}{m}\right) \\ &\leq \text{OPT} + \text{OPT}\left(1 - \frac{1}{m}\right) \\ &= \text{OPT}\left(2 - \frac{1}{m}\right) \end{aligned}$$

Therefore, the upper bound on the ratio of the performance of the greedy solution to the optimal solution is,

$$\frac{T}{\text{OPT}} = \left(2 - \frac{1}{m}\right).$$

□

The following example in Part (d) shows that the above upper bound is tight. It gives a family of examples in which the above greedy algorithm takes $\text{OPT}\left(2 - \frac{1}{m}\right)$ to process m jobs.

(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.

Consider a total of $m^2 - m + 1$ jobs, where the first $m^2 - m$ jobs, each takes 1 minute to process and the last job takes m minutes to process.

The greedy algorithm processes all the 1-minute jobs first, followed by the m-minute job. It assigns m 1-minute jobs to all the m machines first. When they finish, it again assigns the next set of m 1-minute jobs to all the m machines. It does this $(m-1)$ times to process all the $m^2 - m$ one-minute jobs, taking a total of $m - 1$ minutes. It finally assigns the m-minute job to one of the machines. Therefore, the greedy algorithm takes a total of $m - 1 + 2 = 2m - 1$ minutes.

On the other hand, the optimal solution, assigns the m-minute job to one machine first. So, there are $m - 1$ machines left for the $(m^2 - m)$ 1-minute jobs. So, the optimal algorithm, assigns $m - 1$ of the $(m^2 - m)$ jobs to each of the $m - 1$ machines at a time. It takes a total of m minutes to process all of the $(m^2 - m)$ 1-minute jobs on $m - 1$ machines. Therefore, both the m-minute job and the $(m^2 - m)$ 1-minute jobs will all finish in m minutes.

Letting T be the finish time of the greedy algorithm and OPT be the finish time of the optimal algorithm, the factor separating them is therefore,

$$\frac{T}{\text{OPT}} = \left(2 - \frac{1}{m}\right).$$

□