Problem Set 10 Solutions

This problem set is due **at 11:59pm** on **Thursday, December 8, 2016.**

EXERCISES (NOT TO BE TURNED IN)

Competitive Analysis

- Do Problem 17-5 in CLRS on page 476.

Amortized Analysis

- Do Exercise 17.2-3 in CLRS on page 459.
- Do Exercise 17.3-6 in CLRS on page 463.
- Do Exercise 17.3-7 in CLRS on page 463.

NP-Completeness

- Do Exercise 34.2-6 in CLRS on page 1066.
- Do Exercise 34.5-1 in CLRS on page 1100.
- Do Exercise 34.5-3 in CLRS on page 1101.

Approximation Algorithms

- Do Problem 35-4 in CLRS on page 1135.
- Do Problem 35-6 in CLRS on page 1137.

Problem 10-1. Set Using Sorted Lists [20 points]

We want to implement a set data structure which supports the following two operations:

- **INSERT(x)** - add x to the set
- **CONTAINS(x)** - returns true if the set contains x and false otherwise

We will implement the set using sorted lists of length 2^k as follows.

For each value of i from 0 to m, we will maintain a single sorted list L_i. List L_i will either be empty and have 0 elements or it will be full and have 2^i elements. Each individual list is sorted, but there is no relationship between elements in different lists.

At the beginning, $L_0, L_1, \ldots L_m$ are all empty.

We determine which lists should be empty or full depending on the binary representation of how many elements we have inserted so far. For example, if $5 = 1 + 4$ elements have been inserted, then list 1 will be empty and lists 0 and 2 will be full. Below is an example of how such a set might look.

$$L_0 = [6]$$
$$L_1 = [\,]$$
$$L_2 = [2, 5, 8, 9]$$

To implement **INSERT(x)**, we will make a new list of size 1 with the element x in it. If L_0 is empty, we simply replace L_0 with our new list. If L_0 is full, we combine L_0 and our new list into one list of length 2. We then check if L_1 is empty, and if it is, we replace L_1 with our new list of length 2. If L_1 is full, we once again combine L_1 with our new list of length 2 into a length 4 sorted list, and so on.
In the example, if we inserted 3 to the set, the lists would now look like this.

$$L_0 = [\,]$$
$$L_1 = [3, 6]$$
$$L_2 = [2, 5, 8, 9]$$

(a) [2 points] Given this structure, provide an implementation of **CONTAINS(x)** that runs in $O(\log^2 n)$ time, where n is the number of elements in the structure.

Solution: Binary search through each list. There are a total of n INSERT operations, so the total number of elements in the set is at most n when CONTAINS is called. This means there can be at most $O(\log n)$ lists and each list takes at most $O(\log n)$ to binary search through, which gives the desired runtime.

(b) [2 points] While inserting, we frequently need to combine together 2 sorted lists of length 2^p. Name an algorithm that does this in $O(2^p)$ time.

Solution: Use mergesort on the two lists.

(c) [4 points] Let $q = 2^p \cdot c$, where c is odd. Show that the runtime of inserting the q-th element is $O(2^p)$.

Solution: After the insertion, L_p will be full, and all lists below it will be empty. First, we make a list of length 1, then we merge it with another list of length 1, making a list of length 2. We then merge that with a list of length 2, making a list of length 4, and so on. The runtime of this is roughly the runtime of all of the merge operations, which is $1 + 2 + 4 + 8 + \cdots + 2^{p-1} = O(2^p)$.

(d) [4 points] Given a number $p < \log_2 n$, how many times will list L_p go from empty to full during n INSERT operations?

Solution: $\frac{1}{2}\lfloor \frac{n}{2^p} \rfloor$. L_0 becomes full every other iteration, L_1 every 4th, and so on.

(e) [8 points] Use amortized analysis to show that the amortized runtime of INSERT is $O(\log n)$, where n is the number of elements in the structure.

Solution: Combining parts (c) and (d), the total runtime is a sum, for all values of k, of the product of the number of times L_k goes from empty to full and the time it takes to insert an element that will make L_k go from empty to full. This can be written as

$$\sum_k \frac{1}{2}\left\lfloor \frac{n}{2^k} \right\rfloor \cdot O(2^k) = \sum_k O(n) = O(n \log n)$$

Because the total runtime of n operations is $O(n \log n)$, the amortized runtime per operation is $O(\log n)$.

Problem 10-2. Red Line Repairs [40 points]

Your train stop repair company is based in Kendall Square and serves the stops along the Red Line. For this problem, assume that the Red Line is the y-axis, the distance between every two consecutive stops is 1, and Kendall Square is located at $y = 0$.

You have two repairmen who will travel to the stops when they malfunction. Both of your repairmen are able to repair any stop, and if both of them are at the same stop, one of them will arbitrarily be chosen to repair it. They both arrive at Kendall Square at the beginning of the day.

During the day, some stops will suddenly malfunction. When this occurs, at least one of your repairmen must travel to the stop to repair it. Your repairmen must always repair stops in the order that they malfunction, and you won't know the location of the next malfunctioning stop until you repair the current one. Your goal is to minimize the total commute distance of your repairmen.

Ideally, you would like to know the sequence of stops that malfunction ahead of time. Given this information, you could come up with a set of travel instructions for your 2 repairmen that would minimize their total commute distance. Such a strategy would be an *offline* algorithm. We will use D_{off} to denote the minimal commute distance for an offline algorithm given a sequence of malfunctioning stops. We will use $(s_1, s_2, \ldots, s_n)$ to denote the sequence, where each s_i is an integer corresponding to a stop's location.

In reality, you do not know the sequence of malfunctioning stops, so you will come up with an *online* algorithm which is 2-competitive with the offline algorithm

(a) [8 points] You have an idea to minimize total commute distance. When a stop k malfunctions, you will assign whichever of your two repairmen is closer to s_k to repair it. Let D_1 be the total commute distance using this strategy. Show that there exists a sequence of stops malfunctioning such that $D_1 > 100D_{off}$.

Solution: Consider two stops located at $y = 2$ and $y = 3$ that keep malfunctioning over and over 1000 times - that is, the stop malfunction sequence is $2, 3, 2, 3, 2, 3, \ldots$. Your strategy will result in one of your repairmen trying to repair both of these stops, and the total commute distance D_1 will be just over 1000 . However, the best strategy is to send one repairman to repair the stop at $y = 2$ and the other repairman to repair the stop at $y = 3$. Afterwards, they won't have to move when the stops malfunction again. In this strategy, the total commute distance is $D_{off} = 5$. here, $D_1 > 100D_{off}$.

Instead, you will use a different strategy to repair broken stops in an online fashion. Suppose that your repairmen are currently at the locations y_1 and y_2, where $y_1 \leq y_2$, and the next stop that needs to be repaired is stop k.

1. If $y_2 < s_k$, then you will just send your repairman at y_2 to travel to s_k.
2. If $y_1 > s_k$, then you will just send your repairman at y_1 to travel to s_k.
3. If $y_1 < s_k < y_2$, you will ask both repairmen to travel the same distance toward s_k until one of them reaches it. At that point, the other repairman can also stop traveling. For example, if $y_1 = 2, s_k = 3$, and $y_2 = 10$, then you will send your repairman at $y = 2$ to the location $y = 3$ and you will send your repairman at $y = 10$ to the location $y = 9$.

To analyze this online algorithm, we will define f_{i1}, f_{i2} to be the locations of the two repairmen just after repairing malfunctioning stop i in the offline algorithm, where $f_{i1} \leq f_{i2}$. Similarly, define g_{i1}, g_{i2} to be the locations of the two repairmen just after repairing malfunctioning stop i in the online algorithm, where $g_{i1} \leq g_{i2}$.

Then, we will define $\Phi_{a,b} = 2M_{a,b} + d_b$ where

- $M_{a,b} = |f_{a1} - g_{b1}| + |f_{a2} - g_{b2}|$. In other words, it is the distance between the two repairmen with larger y coordinates in the offline and online algorithms plus the distance between the two repairmen with smaller y coordinates in the offline and online algorithms after each algorithm has repaired a specific stop.
- $d_b = g_{b2} - g_{b1}$ is the distance between the repairmen in the online algorithm after repairing malfunctioning stop b

Our ultimate goal is to analyze $\Phi_{i,i}$, which corresponds to when both the offline and online algorithms have repaired the same stop i. We will do so by computing $\Phi_{i,i} - \Phi_{i-1,i-1}$ in two parts. First, we will analyze how the offline algorithm changes Φ and compute $\Phi_{i,i-1} - \Phi_{i-1,i-1}$. Then, we will analyze how the online algorithm changes Φ and compute $\Phi_{i,i} - \Phi_{i,i-1}$.

(b) [8 points] Suppose that the offline algorithm requires your repairmen to travel a total distance of off_i from their locations after repairing malfunctioning stop $i-1$ in order to repair malfunctioning stop i. Show that $\Phi_{i,i-1} - \Phi_{i-1,i-1} \leq 2off_i$

Solution: $\Phi_{i,i-1} - \Phi_{i-1,i-1} = 2(M_{i,i-1} - M_{i-1,i-1}) + d_{i-1} - d_{i-1} = 2(M_{i,i-1} - M_{i-1,i-1})$. Thus, we just have to show $M_{i,i-1} - M_{i-1,i-1} \leq off_i$.

There are two cases. In the first case, the offline repairman with the larger y coordinate at step $i-1$ still has the larger y coordinate at step i. Suppose the amount traveled by this repairman is x. Then, the distance traveled by this repairman increases M by at most x because the term $|f_{i1} - g_{i-1,1}|$ will be at most x bigger than $|f_{i-1,1} - g_{i-1,1}|$, and the distance traveled by the other repairman increases M by at most $off_i - x$ by the same reasoning. Thus, in total, M increases by at most off_i.

In the second case, the offline repairman with the larger y coordinate at step $i-1$ now has the smaller y coordinate at step i. This means that at some time, the two offline repairmen will meet along the y-axis. Suppose that the two repairmen have traveled a distance of z before they meet. At this exact time, the total change to M is at most z by the same reasoning used in the first case. From this point forward, the offline repairman with the larger y-coordinate with continue to have the the larger y-coordinate. Thus, the remaining $off_i - z$ traveled by the two repairmen can be analyzed in the same way as in the first case as well, which means that in total, M will increase by at most off_i.

Thus, $\Phi_{i,i-1} - \Phi_{i-1,i-1} = 2(M_{i,i-1} - M_{i-1,i-1}) \leq 2off_i$.

(c) [12 points] Next, suppose that the online algorithm requires your consultants to travel a total distance of on_i from their locations after repairing malfunctioning stop $i-1$ in order to repair malfunctioning stop i. Show that $\Phi_{i,i} - \Phi_{i,i-1} \leq -on_i$.

Solution: There are two cases.

In the first case, the online algorithm requires that your two repairmen travel toward each other by a distance of $\frac{on_i}{2}$ each until one of them reaches s_i. Note that the offline

algorithm will have moved a repairman to s_i to repair it too. Thus, one of your repairmen in the online algorithm will be getting closer to the location of a repairman in the offline algorithm, so this repairman's movement will actually lower the value of M by $\frac{on_i}{2}$. Thus, even if the other consultant's movement increases the value of M by $\frac{on_i}{2}$, the net effect is that M will not increase.

Finally $d_i - d_{i-1} = -on_i$ because your repairmen get closer to each other. Thus

$$\Phi_{i,i} - \Phi_{i,i-1} = 2(M_{i,i} - M_{i,i-1}) + d_i - d_{i-1} \leq 2 * 0 - on_i = -on_i$$

In the second case, the online algorithm requires exactly one of your two repairmen to travel a distance of on_i toward the client's location. In this case, M will decrease by on_i because your traveling repairman is getting closer to a repairman in the offline algorithm, while your other repairman doesn't move. At the same time, $d_i - d_{i-1} = on_i$ because your two repairmen are getting further away. Thus,

$$\Phi_{i,i} - \Phi_{i,i-1} = 2(M_{i,i} - M_{i,i-1}) + d_i - d_{i-1} = 2 * (-on_i) + on_i = -on_i$$

(d) [12 points] Conclude that the online algorithm is 2-competitive with the offline algorithm over any malfunctioning stop sequence $(s_1, s_2, \ldots, s_n)$.

Solution: Note that $\Phi_{a,b} \geq 0$ always, and also that $\Phi_{0,0} = 0$. We know from combining parts (b) and (c) that $\Phi_{i,i} - \Phi_{i-1,i-1} \leq 2off_i - on_i$. Let D_{on} denote the total commute distance of the online algorithm. Then we have that

$$\Phi_{n,n} = \Phi_{n,n} - \Phi_{0,0} = \sum_{i=1}^{n} \Phi_{i,i} - \Phi_{i-1,i-1} \leq \sum_{i=1}^{n} (2off_i - on_i) = 2D_{off} - D_{on}$$

Thus, $0 \leq \Phi_{n,n} \leq 2D_{off} - D_{on}$, so $D_{on} \leq 2D_{off}$, as desired.

Problem 10-3. Impressing a Venture Capitalist [40 points]

Alyssa P. Hacker is a part-time MIT student who is currently running her own company. Every day, she chooses between working, which increases her wealth, or studying at MIT, which increases her intelligence.

She wants to pitch her company to famed Silicon Valley venture capitalist Monica Hall from Raviga Capital Management. If she impresses Monica with how intelligent she is, Monica will invest in her company and her daily earnings will increase. However, she knows she needs to learn more before she can impress Monica. Finally, she will only have one chance in her life to impress Monica.

Concretely, the problem is as follows. Alyssa starts on day 0 with 0 dollars and 0 intelligence. Every day, she can choose to work, which will earn her 1 dollar, or to study, which will increase

her intelligence by 1. She and Monica will meet on day t_{meet}. If her intelligence is greater than or equal to I_{min} by that day, Monica will invest in her company, increasing her earnings to x dollars per day of work from that day forward, where $x > 1$. If her intelligence is below I_{min}, Monica will not invest in her, and she will continue to earn 1 dollar per day of work. Her goal is to maximize her wealth when she retires on day T.

For the rest of this problem, assume that x, I_{min}, and T are constants that Alyssa knows.

For parts **(a)** and **(b)**, suppose that Alyssa knows the value of t_{meet}, the day she and Monica will meet.

(a) [8 points] Prove that Alyssa's optimal strategy must be one of the following:

1. Work every day for the rest of her life
2. Study until she reaches the intelligence I_{min}, then work for the rest of her life.

Solution: If there exists a strategy where Alyssa works on day t and studies on day $t + 1$, she can swap the order of those two days and it can only improve the final outcome. Thus, her optimal strategy must involve studying for some number of days and then working for some number of days. Studying for fewer than I_{min} days is worse than not studying, and studying for more than I_{min} days is worse than studying for I_{min} days, which leaves us with the two strategies listed.

(b) [8 points] Depending on the values of x, I_{min}, T, t_{meet}, when should she choose strategy 1, and when should she choose strategy 2? How much money will she end up with in each case?

Solution: If she uses strategy 1, she will end up with T dollars.

If she uses strategy 2, she will end up with $T - I_{min}$ dollars if $I_{min} > t_{meet}$, and she will end up with $(t_{meet} - I_{min}) + x(T - t_{meet})$ dollars if $I_{min} \leq t_{meet}$.

Thus, she should use strategy 1 if $I_{min} > t_{meet}$ or if $I_{min} \leq t_{meet}$ and $T > (t_{meet} - I_{min}) + x(T - t_{meet})$, and she should use strategy 2 otherwise.

Now we will assume that Alyssa does not know the value of t_{meet}. The strategy from part **(a)** is an *offline* algorithm for this problem. We will help Alyssa devise an *online* algorithm to maximize her wealth.

It turns out that the optimal online algorithm for Alyssa also falls into one of the following two categories:

1. Work every day for the rest of her life
2. Study until one of the following two occurs.
   (a) She reaches I_{min} intelligence

(b) Monica arrives.

After that, Alyssa will work for the rest of her life. Thus, if Monica arrives before Alyssa reaches intelligence I_{min}, Alyssa will stop studying.

(c) [8 points] Suppose that Alyssa decides to use strategy 1 as her online algorithm. Prove that this algorithm is at least $\min(1, 1/(x(1 - \frac{I_{min}}{T})))$-competitive.

Note: The *competitive ratio* here is defined as the ratio of the amount of money Alyssa ends up with using her online algorithm and the amount of money Alyssa ends up with using her offline algorithm.

Solution: Her strategy always gets T dollars. We are given that the optimal strategy is either strategy 1 or strategy 2. If the optimal strategy is strategy 1, then the competitive ratio is 1. Otherwise, if the optimal strategy is strategy 2, this strategy gets $t_{meet} - I_{min}$ dollars from the work done before Monica arrives and $x(T - t_{meet})$ dollars from the work done after. So the competitive ratio is

$$\begin{aligned}
\frac{T}{t_{meet} - I_{min} + x(T - t_{meet})} &= \frac{1}{\frac{t_{meet} - I_{min}}{T} + x(1 - \frac{t_{meet}}{T})} \\
&\geq \frac{1}{x\left(\frac{t_{meet} - I_{min}}{T}\right) + x(1 - \frac{t_{meet}}{T})} \\
&= \frac{1}{x\left(\frac{t_{meet} - I_{min}}{T} + 1 - \frac{t_{meet}}{T}\right)} \\
&= \frac{1}{x\left(1 - \frac{I_{min}}{T}\right)}
\end{aligned}$$

where we have used the fact that $x > 1$ and also that (in this case) $I_{min} \leq t_{meet}$.

(d) [8 points] Suppose that Alyssa decides to use strategy 2 as her online algorithm. Prove that this algorithm is at least $\left(1 - \frac{I_{min}}{T}\right)$-competitive.

Solution: If the optimal strategy is strategy 2, the competitive ratio is 1. If the optimal strategy is strategy 2, the competitive ratio is $\frac{T - t_{meet}}{T}$. In the second case, we know that $I_{min} > t_{meet}$. Finally, we know that $\frac{T - t_{meet}}{T} > 1 - \frac{t_{meet}}{T} > 1 - \frac{I_{min}}{T}$

(e) [8 points] Find an online algorithm that achieves a competitive ratio of $1/\sqrt{x}$ regardless of the values of t_{meet} and T.

Solution: Depending on the values of I_{min} and T (specifically, the ratio $\frac{I_{min}}{T}$), we will choose the better of strategy 1 and strategy 2 to be our online algorithm. We will end

up a competitive ratio of $\max\left(\frac{1}{x(1-\frac{I_{min}}{T})}, 1-\frac{I_{min}}{T}\right)$. The two quantities are inversely proportional, so the worst case is when they are equal. At that point, the competitive ratio will be exactly $1/\sqrt{x}$.

Problem 10-4. Matrix Fixing [40 points]

You are given an $n \times n$ matrix M of integers. To modify the matrix, you may repeatedly perform the following operation: pick an integer i and set all the entries from the i-th row *and* the i-th column to zero (a total of $2n-1$ entries). Given M, your goal is to use the minimum number of these operations to turn M into a matrix of all zeros. We'll call this the **matrix-fixing problem**.

(a) [5 points] State the matrix-fixing problem as a decision problem.

Solution: Given an $n \times n$ matrix M of integers and an integer k, decide if it is possible to turn M into a matrix of all zeros using at most k operations.

(b) [10 points] Show that the decision problem is in NP.

Solution: Given M, k, and a solution (a certificate containing a list of up to k operations), we can verify that (M, k) is a valid YES input by:

- Making sure that the solution uses at most k operations.
- Making sure that after applying those k operations, M is the zero matrix.

The certificate here has polynomial size because it consists of k integers. It takes $O(n)$ to apply each operation, and there are at most $O(n)$ operations in any solution. Then, it takes $O(n^2)$ to check whether M becomes the zero matrix. Overall, this verification algorithm uses polynomial time. Because there exists a polynomial-time algorithm to verify any YES input given a certificate, the matrix-fixing problem is in NP.

(c) [25 points] Show that the decision problem is NP-complete.
Hint: You may use the fact that the decision variant of the clique problem is NP-complete.

Solution: The clique problem is: given a graph $G = (V, E)$ and an integer k, is there a subset of at least k vertices that are fully connected to each other? We will show that the matrix-fixing problem is NP-hard by giving a reduction from the clique problem to the matrix-fixing problem:

1. Given an input $x = (G, k)$ to the clique problem, create A to be the opposite of the adjacency matrix of G: $A_{i,j}$ is 0 if there is an edge between vertices i and j and 1 otherwise. The corresponding input to the matrix-fixing problem is $(A, |V| - k)$, where A is the matrix to fix and $|V| - k$ is the number of operations we are

allowed. Creating A takes at most $O(n^2)$ time, so this reduction takes polynomial time. To prove this reduction works, we'll show that an instance of the clique problem is a YES instance iff it reduces to a YES instance of the matrix-fixing problem.

2. First, we'll show that a YES instance of the clique problem will reduce to a YES instance of the matrix-fixing problem. If G had a clique of size at least k, then removing the rows and columns corresponding to the vertices that are not part of this clique (of which there are at most $|V| - k$) will leave us with a submatrix of A such that all of its entries are 0. This process corresponds exactly to running the operation on the $|V| - k$ indices corresponding to the non-clique vertices and being left with only 0s, and so $(A, |V| - k)$ would be a YES instance of the matrix-fixing problem.
3. Now, suppose that a clique problem instance reduced to a YES instance of the matrix-fixing problem. We'll show that the clique problem instance must also have been a YES. If there exists a set of at most $|V| - k$ indices such that running operations on these indices left you with all 0s, then consider what happens when you remove the vertices corresponding to these indices from G. At least k vertices will remain in V and their adjacency matrix corresponds to the subsection of A that didn't have operations performed on it. We know that this subsection must have initially been all 0s, so this subset of at least k vertices from V are fully connected and form a clique. This makes (G, k) a YES instance of the clique problem.
4. Because an instance of the clique problem is YES iff it reduces to a YES instance of the matrix-fixing problem, our reduction is valid.

Because the matrix-fixing problem is both NP-hard and in NP, the matrix-fixing problem is NP-complete.

Alternate solution (reduction from vertex cover):

The vertex cover problem is: given a graph $G = (V, E)$ and an integer k, is there a subset of at most k vertices such that every edge touches at least one of those vertices? We will show that the matrix-fixing problem is NP-hard by giving a reduction from the vertex problem to the matrix-fixing problem:

1. Given an input $x = (G, k)$ to the vertex cover problem, create A to be the adjacency matrix of G. The corresponding input to the matrix-fixing problem is (A, k), where A is the matrix to fix and k is the number of operations we are allowed. Creating A takes at most $O(n^2)$ time, so this reduction takes polynomial time. To prove this reduction works, we'll show that an instance of the vertex cover problem is a YES instance iff it reduces to a YES instance of the matrix-fixing problem.
2. First, we'll show that a YES instance of the vertex cover problem will reduce to a YES instance of the matrix-fixing problem. If G has a vertex cover of size at most

k, then we can perform the k operations at the indices corresponding to the vertices of the vertex cover. Each edge in G touches one of those k vertices, so each 1 in the adjacency matrix must be in one of the columns or rows corresponding to those vertices. This means that performing the corresponding set of operations will hit all the 1s in A. Thus, (A, k) will be a YES instance of the matrix-fixing problem.

3. Now, suppose that a vertex cover instance reduced to a YES instance of the matrix-fixing problem. We'll show that the vertex cover instance must have been a YES. If there were a set of k operations that set all the 1s in the adjacency matrix to 0, then the set of vertices corresponding to those k operations form a vertex cover. Because all the 1s in A must have been in one of the rows/columns that had an operation, we know that every edge in G touches one of the vertices we picked for our vertex cover. Since we found a vertex cover of at most k vertices, (G, k) must be a YES instance of vertex cover.
4. Because an instance of the vertex cover problem is YES iff it reduces to a YES instance of the matrix-fixing problem, our reduction is valid.

Because the matrix-fixing problem is both NP-hard and in NP, the matrix-fixing problem is NP-complete.

Now, we'll focus on the optimization variant of the matrix-fixing problem. Consider the following greedy approximation algorithm: as long as there are still nonzero entries in M, perform the operation that converts the maximum number of nonzero entries to zero. In the remaining parts of this problem, you will prove that this greedy algorithm always gives an approximation ratio of $2 \ln n$. That is, if the optimal solution uses k operations, then the greedy algorithm will use no more than $2k \ln n$ operations.

The following two parts are extra credit. We recommend trying them since they are useful preparation for the final exam.

(d) [5 points] **(Bonus)** Let k be the number of operations used by the optimal solution to the matrix-fixing problem. Show that there must exist an operation on M that changes at least a $1/k$ fraction of the nonzero entries in M to zero.

Hint: Prove by contradiction.

Solution: We can prove this by contradiction. Suppose that all k operations in the optimal solution change less than $1/k$ fraction of the nonzero entries in M. Let m be the initial number of nonzero entries. Combined, the k operations will change less than $k * (1/k) * m = m$ of the nonzero entries to zero. This gives a contradiction, because in order for the operations to be a valid solution, they must have changed all m of the nonzero entries to 0.

(e) [10 points] **(Bonus)** Using what you just proved, show that the greedy approximation algorithm will use no more than $2k \ln n$ operations.

Hint: You may find the following fact useful: $(1 - m) < e^{-m}$ for all positive m.

Solution: Let m be the initial number of nonzero entries. After the first greedily chosen operation, we'll change at least a $1/k$ fraction of the nonzero entries to zero, and we'll be left with $m(1 - 1/k)$ nonzero entries.

Because the original M had a size-k solution, this new matrix must also have a size-k solution, given that its nonzero entries are a subset of the original nonzero entries. Using the same logic as before, we know that some operation exists that will change at least a $1/k$ fraction of the remaining nonzero entries to zero. This leaves us with $m(1 - 1/k)^2$ nonzero entries.

Continuing this, after $2k \ln n$ operations, the number of nonzero entries remaining is at most $m(1 - 1/k)^{2k \ln n} < me^{-(2k \ln n)/k} = me^{-2 \ln n} = m/n^2 \leq 1$. This tells us that after $2k \ln n$ greedily chosen operations, less than 1 nonzero entry remains and we have turned M into all zeros.

Problem 10-5. L-Cover [40 points]

An L-polygon is a polygon in the shape of an L, such as those in Figure 1. More formally, a size s L-polygon is any polygon created by cutting an $(s-1) \times (s-1)$ square out of the top right corner of an axis-aligned $s \times s$ square.

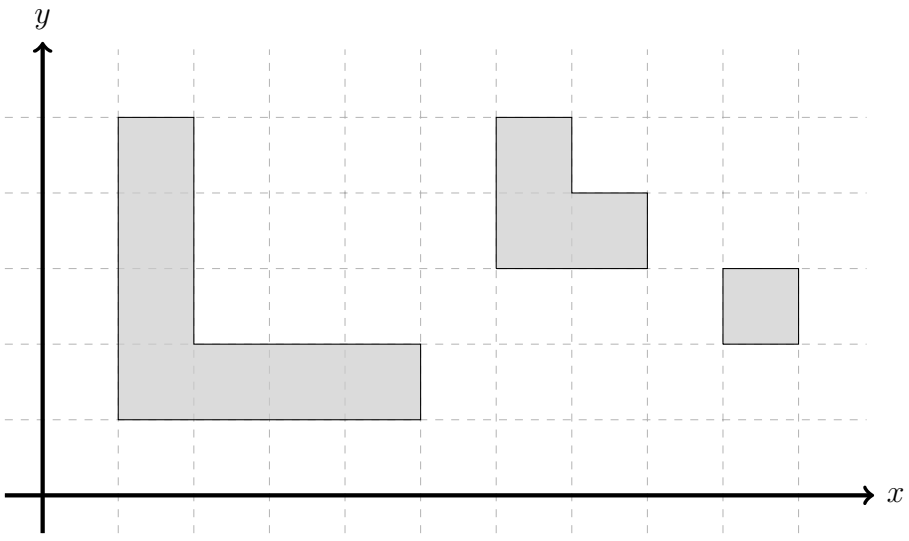


Figure 1: Examples of valid L-polygons, with sizes 4, 2, and 1. The rightmost L-polygon is the smallest possible L-polygon.

In the **L-cover problem**, you are given the coordinates of a set P of n points and the coordinates of a set L of L-polygons, each with the same size s. Your goal is to find the minimum number of

L-polygons from L needed to cover all n points. A point is covered by an L-polygon if it lies in the interior or on an edge of the L-polygon. An example of the L-cover problem is shown in Figure 2.

(a) [5 points] State the L-cover problem as a decision problem.

Solution: Given a set P of n points, a set L of L-polygons, and an integer k, decide whether it is possible for k L-polygons from L to cover all n points.

(b) [10 points] Show that the decision problem is in NP.

Solution: Given P, L, k, and a solution (a certificate containing a list of up to k L-polygons from L), we can verify that (P, L, k) is a YES instance by:

- Making sure that the solution uses no more than k L-polygons.
- Making sure that every point is covered by at least one of the L-polygons in the solution.

The certificate here has polynomial size because it is a subset of the input. There are $O(n)$ points to check, and for each point, we can iterate through the set of L-polygons in the solution. For each L-polygon, we can check whether the point is inside the L-polygon in constant time. Overall, this verification algorithm uses time polynomial in the input size. Because there exists a polynomial-time algorithm to verify any YES input given a certificate, the L-polygon problem is in NP.

(c) [25 points] Show that the decision problem is NP-complete.

Hint: You may use the fact that the decision variant of the vertex cover problem is NP-complete.

Hint: If you reduce from vertex cover, think about adjacency matrices. Consider visualizing adjacency matrices on a grid, as shown in Figure 3.

Solution: The vertex cover problem is: given a graph $G = (V, E)$ and an integer k, is there a subset of at most k vertices such that every edge touches at least one of those vertices? We will show that the L-cover problem is NP-hard by giving a reduction from the vertex problem to the L-cover problem:

1. Given an input $x = (G, k)$ to the vertex cover problem, create A to be the adjacency matrix of G. We will assume that the vertices are indexed 1 through $|V|$. Also, let $n = |V|$. To create an input for the L-cover problem, do the following:
   - Ignore the half of A that is on or below the main diagonal.
   - For each entry of the top half of A, if $A_{i,j} = 1$, add a point at $(j + 0.5, (n - i + 1) + 0.5)$. (This is equivalent drawing a dot at every 1 in the adjacency matrix upper half in Figure 3.)

- For each vertex i, add a size n L-polygon with its bottom left corner at $(i, n - i + 1)$. This is equivalent to creating an L-polygon that covers all the entries in the upper half of the adjacency matrix that are part of the ith row or the ith column. (In Figure 3, we'd place L-polygons with bottom-left corners at the bottom-left corners of (A, A), (B, B), (C, C), (D, D), (E, E), and (F, F).)
- Use the same k for the L-cover problem.
- This reduction takes polynomial time because the number of points is at most the number of edges in G and the number of L-polygons is at most the number of vertices in G.

To prove that this reduction works, we'll show that an instance of the vertex cover problem is a YES instance iff it reduces to a YES instance of the L-cover problem.

2. First, we'll show that a YES instance of the vertex-cover problem will reduce to a YES instance of the L-cover problem. Given the vertex cover of at most size k, we can cover all the points in the L-cover input by using the exact L-polygons that correspond to the vertices of the vertex cover. The L-polygon corresponding to a vertex i will cover all the points corresponding to edges incident to vertex i. Because the vertices of the vertex cover covered all the edges, this set of L-polygons will cover all the points. This makes our L-cover problem a YES instance.
3. Now, suppose that a vertex cover instance reduced to a YES instance of the L-cover problem. We'll show that the vertex cover instance must have been a YES. Using similar reasoning, consider the set of k L-polygons used to cover all the points. Take the corresponding k vertices as the vertex cover. The L-polygons collectively covered all the points, and the points corresponded to edges. As a result, the corresponding vertices will cover the same edges, which is all the edges. This means that (G, k) was a YES instance of vertex cover.
4. Because YES instances from the vertex cover problem correspond to YES instances of the L-cover problems, our reduction is valid.

Because the L-cover problem is both NP-hard and in NP, the L-cover problem is NP-complete.

Now, we'll focus on the optimization variant of the L-cover problem. Consider the following greedy approximation algorithm: as long as there are points that are not covered, pick the L-polygon from L that covers the most of these uncovered points. In the remaining parts of this problem, you will prove that this greedy algorithm always gives an approximation ratio of $\ln n$. That is, if the optimal solution uses k L-polygons, then the greedy algorithm will use no more than $k \ln n$ L-polygons.

The following two parts are extra credit. We recommend trying them since they are useful preparation for the final exam.

(d) [5 points] **(Bonus)** Let k be the number of L-polygons used by the optimal solution

to the L-cover problem. Show that there must exist an L-polygon in L that covers at least a $1/k$ fraction of the points in P.

Hint: Prove by contradiction.

Solution: We can prove this by contradiction. Suppose that all k L-polygons in the optimal solution cover less than a $1/k$ fraction of the points in P. Combined, the k L-polygons will cover less than $k*(1/k)*n = n$ of the points. This gives a contradiction, because in order for the L-polygons to be a valid solution, they must have covered all n of the points.

(e) [10 points] **(Bonus)** Using what you just proved, show that the greedy approximation algorithm will use no more than $k \ln n$ L-polygons.

Hint: You may find the following fact useful: $(1 - m) < e^{-m}$ for all positive m.

Solution: After the first greedily chosen L-polygon, we'll cover at least $1/k$ fraction of the L-polygons. We'll be left with $n(1 - 1/k)$ uncovered points at most.

Because the original set of points had a size-k L-cover, the set of remaining points also has a size-k L-cover. Using the same logic as before, we know that some L-polygon exists that will cover at least $1/k$ fraction of the remaining points, leaving us with $n(1 - 1/k)^2$ uncovered points.

After $k \ln n$ operations, the number of uncovered points remaining is at most $n(1 - 1/k)^{k \ln n} < ne^{-(k \ln n)/k} = ne^{-\ln n} = 1$. This tells us that after $2k \ln n$ greedily chosen L-polygons, less than 1 uncovered point remains (all points are covered).

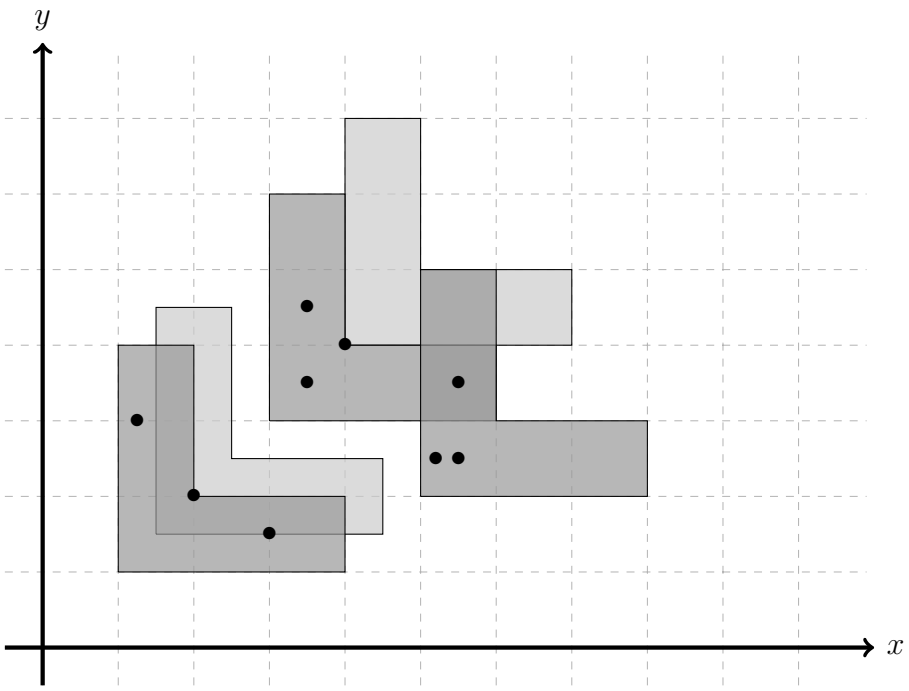
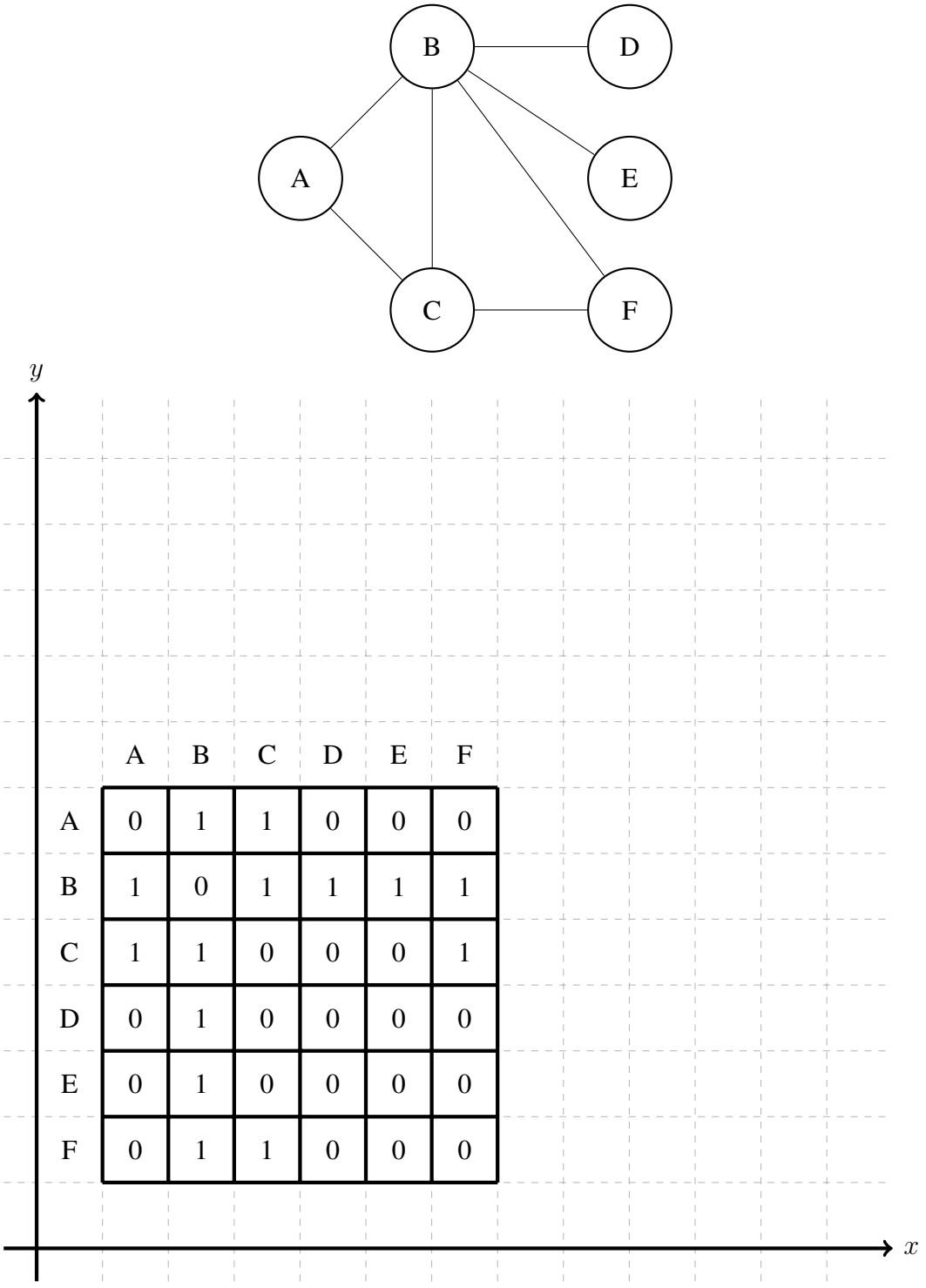


Figure 2: An example input to the L-cover problem. There are five L-polygons and nine points, and each L-polygon has size three. The minimum number of these L-polygons needed to cover all the points is three (shown in the darker color). You may not move the L-polygons away from their initial positions.



	A	B	C	D	E	F
A	0	1	1	0	0	0
B	1	0	1	1	1	1
C	1	1	0	0	0	1
D	0	1	0	0	0	0
E	0	1	0	0	0	0
F	0	1	1	0	0	0

Figure 3: An example of the hint for problem 5(c). The adjacency matrix for the above example graph is placed directly on a grid. Visualizing a graph this way might help you find a reduction.