CS 170 Efficient Algorithms and Intractable Problems
Spring 2017 Prasad Raghavendra and Sanjam Garg

Homework 2

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

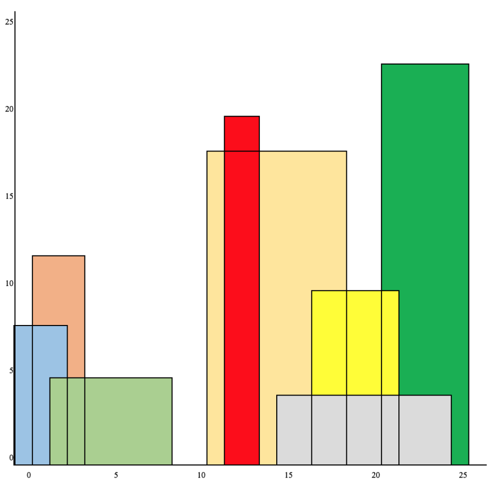
- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

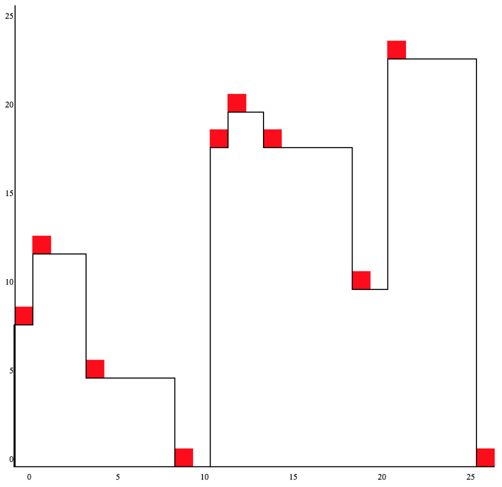
Due Tuesday, February 7, at 11:59am

1. (★★★★ level) Million Dollar View

A \$70 million penthouse has recently been listed for sale in San Francisco. What used to be a perfect view of the Golden Gate Bridge has been marred by a recent development of towers and tech office buildings. Your job is to determine if the penthouse is worth the price, by obtaining the skyscrapers' outline.

Suppose you are given the left and right positions of each of the buildings as well as the height in an array A i.e. $[(lt_1, rt_1, ht_1), (lt_2, rt_2, ht_2), \cdots, (lt_n, rt_n, ht_n)]$. The left and right positions are on the x-axis and the height is on y-axis. The output will be the key coordinates of the outline. These key coordinates are left points of the horizontal lines that form the outline. Thus we want the left point of the line as the x coordinate and the height as the y coordinate, i.e. $[(lt1, ht1), (lt2, ht2)...]$. Input building positions are unsorted. See below for an example.





Input buildings: $[(0,3,8), (1,4,12), (2,9,5), (11,19,18), (12,14,20), (15,25,4), (17,22,10), (21,26,23)]$
Output blocks: $[(0,8), (1,12), (4,5), (9,0), (11,18), (12,20), (14,18), (19,10), (21,23), (26,0)]$
Design an efficient algorithm to compute the outline.
Assume that the base of buildings can get arbitrarily larger than n.
$W = \max(rt_i - lt_i) \quad \forall i \in \{1, \ldots, n\}, W \geq n.$

Solution:

(i) **Main idea** Partition the array into two halves and recursively compute the skyline for each half. Merge and sort the two skylines by taking the smallest left coordinate of the two skylines as a key point. We keep h_1 and h_2 as the height of the last block used from skyline 1 and skyline 2 respectively. The height of the key point will be $\max(h_1, h_2)$.

(ii) **Psuedocode**

```
procedure SKYLINE($A[b_1..b_n]$)
    if $n = 1$ then                                        ▷ base case
        Return [(LEFT(A[1]), HEIGHT(A[1])), (RIGHT(A[1]), 0)].
    $m \leftarrow \lceil n/2 \rceil$                                     ▷ divide and conquer
    $B \leftarrow$ SKYLINE($A[1..m]$)
    $C \leftarrow$ SKYLINE($A[m+1..n]$)
    $S \leftarrow$ empty list
    $h_1 \leftarrow 0$
```

```
$h_2 \leftarrow 0$
while $B$ is not empty and $C$ is not empty do
    if $C$ is empty or LEFT($B[1]$) $<$ LEFT($C[1]$) then
        $x \leftarrow$ LEFT($B[1]$)
        $h_1 \leftarrow$ HEIGHT($B[1]$)
        $h \leftarrow$ MAX($h_1, h_2$)
        Remove $B[1]$ from $B$
    else if $B$ is empty or LEFT($C[1]$) $<$ LEFT($B[1]$) then
        $x \leftarrow$ LEFT($C[1]$)
        $h_2 \leftarrow$ HEIGHT($C[1]$)
        $h \leftarrow$ MAX($h_1, h_2$)
        Remove $C[1]$ from $C$
    else
        $x \leftarrow$ LEFT($C[1]$)
        $h_1 \leftarrow$ HEIGHT($B[1]$)
        $h_2 \leftarrow$ HEIGHT($C[1]$)
        $h \leftarrow$ MAX($h_1, h_2$)
        Remove $B[1]$ from $B$
        Remove $C[1]$ from $C$
    if LENGTH(S) = 0 or $h \neq$ height of last added block then
        Append $(x, h)$ to $S$
return $S$
```

(iii) **Proof of correctness**

- **Base Case**
  If A only has length of 1, then the key point will be the left corner of the building (left and height) and where it ends (right and height as 0).
- **Inductive Hypothesis**
  Assume that the procedure is correct with $n = 1 \cdots k$.
- **Inductive Step**
  Suppose that array A has length $n = k + 1$ can be split into two sub-arrays. Via the inductive hypothesis, we have the skyline blocks from the two sub-arrays S_1, S_2. Since the two skylines are sorted by left coordinate, comparing the first block of S_1, S_2 guarantees us the smallest left coordinate to be used as the next block to be added. If we used the first block of S_1, we update h_1, and likewise for S_2 and h_2. When adding the next critical point, the height will be the higher of h_1 and h_2. WLOG, suppose we are looking to add the next point (l, h) from S_1. h_1 is updated to h. The new point that we add will be $\max(h, h_2)$. We do not need to check previous h_1 because via the inductive hypothesis, we know that this point will not be affected by another point from the same outline. If $h_2 > h$, then we use h_2, meaning that there is a key point from S_2 with a smaller left coordinate that dominates this current one (l, h). Because we update h_2 when we look at a point from S_2, h_2 could get lower in some later comparison, indicating the skyline from S_2 has dropped. If $h_2 < h$, then we use h as this key critical point starts a new portion of the merged skyline Since we only append the key point if the height is not the same as the previous key point, there will not be redundant key points. This merge will be completed until all blocks are S_1 and S_2 are used to complete our resulting skyline.

(iv) **Running time analysis** At each partition, we go through all the skylines in $O(n)$. We have the following recurrence relation $T(n) = 2T(n/2) + O(n)$ and by the master theorem, the running time is

$O(n \log n)$.

Alternate Solution

(a) **Main Idea**
Go through all buildings and add the next key coordinate as the x-coordinate point and the max height of active buildings. We do this by first splitting each of the input buildings into left and right points (l_i, i) and (r_i, i). We then go through each of these critical points and add a building into a max binary heap to keep track of the maximum height among active buildings. We use pointers to the heap so that we we reach a right point, we remove the building from the heap.

(b) **Psuedocode**

```
procedure SKYLINE($A[b_1..b_n]$)
    $S \leftarrow$ empty list
    $B \leftarrow [\text{NULL}_1..\text{NULL}_n]$
    $C \leftarrow \text{SORT}([(l_i,i),(r_i,i)]), \quad i \in \{1,\dots,n\}$
    $H \leftarrow$ empty MAXBINARYHEAP
    for $(x,i)$ in $C$ do
        if $B[i]$ is not NULL then
            HEAPINSERT($H$, (HEIGHT($A[i]$), $i$))
            $B[i] \leftarrow$ pointer to $i$ in heap
        else
            HEAPREMOVE($H$, $B[i]$)
        if $H$ is not empty then
            $h \leftarrow$ HEAPMAX($H$)
        else
            $h \leftarrow 0$
        if LENGTH($S$) $= 0$ or $h$ is not same height as last added element then
            Append $(x,h)$ to $S$
            if $x$ same as last added x coordinate then
                Use larger $h$
    return $S$
```

(c) **Proof of Correctness**

- **Base Case**
  If there is only one building, then when we insert the left point into the heap, we have (l,h) as a key point as the heap only has h and would be the max. When we look at the next point, we know that it is the right coordinate of the building as there is pointer in B to the heap index. Thus we remove the element from the heap and add $(r,0)$ as a key point as the heap would be empty.
- **Inductive Hypothesis**
  Assume that we have already looked at some m point from the $2n$ points after the split and the algorithm correctly adds the key points.
- **Inductive Step**
  Suppose we look at the next element (x,i). Suppose $B[i]$ is NULL. x would be the left coordinate of a new building. We insert this into the heap because we want to consider its height when looking at the maximum element from the heap. The heap will contain the heights of all active buildings, those whose right coordinate we have not visited. The key point will have the height of the maximum value from the heap as this height is the dominating height among all active buildings. Now suppose $B[i]$ is not NULL. x is the right coordinate of building that exist in the

heap. We remove this element from the heap as we do not want this height to be dominating any other buildings that follow. Again the key point for x will use the maximum height that exist in the heap as there could be an active building with a height that at x that forms the skyline. In either case, we only add the key coordinate if the last added coordinate does not have the same height. This would prevent adding redundant points. Since the previous m elements are correct via the hypothesis and this this new coordinate is also correct. The algorithm is correct for all $2n$ pairs.

(d) **Running time analysis**
HEAPINSERT and HEAPREMOVE takes $O(\log n)$ with a binary heap. HEAPMAX peaks at the top element in constant time. We have $2n$ iterations and thus the final running time is $O(n \log n)$.

2. (★★★ level) Majority Elements

An array $A[1 \ldots n]$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be **no** comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) However you *can* answer questions of the form: "is $A[i] = A[j]$?" in constant time.

Can you give a linear-time algorithm?

(Hint: Here's a divide-and-conquer approach:

- Pair up the elements of A arbitrarily, to get about $n/2$ pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them
- If $|A|$ is odd, there will be one unpaired element. What should you do with this element?

Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if A does.)

Note: for this problem, assume you don't have access to a good hash function – i.e. perhaps it takes $O(n)$ to hash an item. In the real world, this is usually not a restriction you need to deal with; but for the pedagogical purposes of this problem assume you can only check for equality of the items quickly.

Solution:

(i) **Main idea** Removing two different items does not change the majority element (but may create spurious solutions!). We recursively pair up the elements and remove all pairs with different elements. For every pair with identical elements, it suffices to keep just one (since we do so for all pairs, we again don't change the majority). If n is odd, brute-force test in linear time whether the unpaired element is a majority element.

(ii) **Psuedocode**

```
procedure MAJORITY-PAIRS($A[1..n]$)
    if $n = 1$ then                                ▷ base case
        Return $A[1]$.

    if $n$ is odd then
        $c \leftarrow 0$                           ▷ is $A[n]$ a majority element?
        for $i \in 1..n-1$ do
            if $A[i] == A[n]$ then $c++$.
        if $c \geq (n-1)/2$ then
            Return $A[n]$.
        else
            Return MAJORITY-PAIRS($A[1..n-1]$)            ▷ $A[n]$ is not a majority element

    else                                                    ▷ even $n$
        $j \leftarrow 1$
        for $i \in 1..n/2$ do
            if $A[2i] == A[2i+1]$ then
                $B[j] \leftarrow A[2i]$
                $j{+}{+}$
        $v \leftarrow$ MAJORITY-PAIRS($B[1..j-1]$)
        $c \leftarrow 0$                                    ▷ is $v$ really a majority element?
        for $i \in 1..n$ do
            if $A[i] == v$ then $c{+}{+}$.
        if $c > n/2$ then
            Return $v$.
        else
            Return $\perp$                                  ▷ there is no majority element
```

(iii) **Proof of correctness** By induction:

Induction hypothesis The algorithm is correct for smaller inputs.

Base Case If $n = 1$, the array contains exactly one element, and we always return it.

Induction step If n is odd, the brute-force test for $A[n]$ is straightforward.

Assume wlog that n is even. We must show that if A has a majority element, then it is also a majority element in B; then we'd be done by the induction hypothesis. Consider a modified algorithm which only removes all the pairs with different elements; let's call the resulting array B'. Suppose v appears in A more often than all other elements combined. Whenever we remove a pair that contains v and another element, this property continues to hold. When we remove a pair that contains two other elements, v definitely remains a majority element. Either way, after each pair removal v remains a majority element. Therefore v (by another induction) is also a majority element in B'. Now from B' to B we do not change the proportions of elements, so the majority elements remains.

(iv) **Running time analysis** One call to a problem of size $\leq n/2$, as well as additional linear time to test each candidate: $T(n) = T(\frac{n}{2}) + O(n) = O(n)$.

3. (★★★ level) Triple Sum

Design an efficient algorithm for the following problem. We are given an array $A[0..n-1]$ with n elements, where each element of A is an integer in the range $0 \leq A[i] \leq 10n$. We are also given an integer t. The algorithm must answer the following yes-or-no question: do there exist indices i, j, k such that $A[i] + A[j] + A[k] = t$?

Design an $O(n \log n)$ time algorithm for this problem.

Hint: define a polynomial of degree $O(n)$ based upon A, then use FFT for fast polynomial multiplication.

Solution:

Main idea: Exponentiation converts addition to multiplication. So, define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether $p(x)^3$ contains x^t as a term.

Pseudocode:

```
Algorithm TripleSum($A[0\ldots n-1]$, $t$):
1. Set $p(x) := \sum_{i=0}^{n-1} x^{A[i]}$.
2. Set $q(x) := p(x) \cdot p(x) \cdot p(x)$, computed using the FFT.
3. Return whether the coefficient of $x^t$ in $q$ is nonzero.
```

Correctness: Observe that

$$q(x) = p(x)^3 = \left(\sum_{0 \le i < n} x^{A[i]}\right)^3 = \sum_{0 \le i,j,k < n} x^{A[i]}x^{A[j]}x^{A[k]} = \sum_{0 \le i,j,k < n} x^{A[i]+A[j]+A[k]}.$$

Therefore, the coefficient of x^t in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] = t$. So the algorithm is correct. (In fact, it does more: the coefficient of x^t tells us *how many* such triples (i, j, k) there are.)

Constructing $p(x)$ clearly takes $O(n)$ time. Since $0 \le A[i] \le 10n$, $p(x)$ is a polynomial of degree at most $10n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^t takes constant time, so overall the algorithm takes $O(n \log n)$ time.

Comment: This problem promised you that each element of the array is in the range $0 \ldots 10n$. What if we didn't have any such promise? Then the FFT-based method above becomes inefficient (because the degree of the polynomial is as large as the largest element of A). It is easy to find a $O(n^2)$ time algorithm, but no faster algorithm is known. In particular, it is a famous open problem (called the 3SUM problem) whether this problem can be solved more efficiently than $O(n^2)$ time. This problem has been studied extensively, because it is closely connected to a number of problems in computational geometry.

Comment: The technique used to solve this problem (namely encoding information in the coefficients of a polynomial and then manipulating the polynomial in some way) is closely related to the technique of generating functions, which are used in combinatorics to do many cool things such as giving a standard way to find a closed form solution to certain types of recurrence relations (for instance the recurrence relation defining the Fibonacci numbers).

4. (★★★★ level) Shortcut Question: Sherlock

Sherlock Holmes is trying to write a computer antivirus program. He starts by modeling his problem. He thinks of computer RAM as being a binary string s_2 of length m. He thinks of a virus as being a binary string s_1 of length $n < m$. His program needs to find all occurrences of s_1 in s_2 in order to get rid of the virus. Even worse, though, these viruses are still damaging if they differ slightly from s_1. So he wants to find all copies of s_1 in s_2 that differ in at most k locations for arbitrary $k \le n$.

(a) Find an $O(nm)$ algorithm for this problem. Give only the main idea; no four-part solution necessary.

Solution: Try all $m - n + 1$ possible starting locations for s_1 in s_2 and for each count the mismatched bits in $O(n)$ time, yielding the location if there are k or fewer mismatched bits. Over all we get $O(n(m-n)) = O(nm)$. Since we try all possible starting locations for s_1, we can be sure that we will find an occurrence if there is any.

(b) Find polynomials p_1 of degree $n-1$ and p_2 of degree $m-1$ such that the coefficient of x^{n-1+i} of $p_1 p_2$ is $n - 2u(i)$ where $u(i)$ is the number of mismatched bits between s_1 shifted by i and s_2. Show how to generate p_1 and p_2 in $O(n+m)$ time.

Solution: Let $p_1(x) = \sum_{i=0}^{n-1} b_i x^{n-1-i}$ and $p_2(x) = \sum_{i=0}^{m-1} c_i x^i$ where $b_i = 1$ if $s_1(i) = 0$ and $b_i = -1$ if $s_1(i) = 1$ and $c_i = 1$ if $s_2(i) = 0$ and $c_i = -1$ if $s_s(i) = 1$. Now the coefficient d_{n-1+i} of x^{n-1+i} in $p_1 p_2$ is:

$$\sum_{j=0}^{n-1} b_{n-1-j} c_{i+j}$$

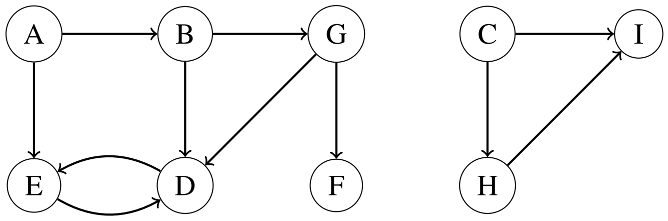
And $b_{n-1-j} c_{i+j} = 1$ if $s_1(j)$ and $s_2(i+j)$ agree and is -1 otherwise. Thus $d_{n-1+i} = n - 2u(i)$ as needed. It is clear how to generate p_1 and p_2 from s_1 and s_2 in order $O(n+m)$.

(c) Give the main idea and runtime for an $O(m \log m)$ algorithm for solving the problem for any k. Again, no four-part solution is necessary, but please clearly describe the algorithm and give pseudocode if you think it will enhance our understanding of your solution.

Solution: Generate p_1 and p_2 $O(n+m)$ using part 2. Then use FFT to multiply the two polynomials in $O(m \log m)$ retrieving all the coefficients to the polynomial product. Then read the overlaps in $O(m)$, and yield a location where the coefficient is $n - 2k$ or higher. The overall runtime is dominated by the polynomial multiplication step which takes $O(m \log m)$.

5. (★★★ level) Graph Basics

For parts (a) and (b), refer to the figure below. For parts (c) through (f), please prove only for simple graphs; that is, graphs that do not have any parallel edges or self-loops.



(a) Run DFS at node A, trying to visit nodes alphabetically (e.g. given a choice between nodes D and F, visit D first).

- List the nodes in the order you visit them (so each node should appear in the ordering exactly once).
- List each node with its pre- and post-number. The numbering starts from 1 and ends at 18.
- Label each edge as **T**ree, **B**ack, **F**orward or **C**ross.

Solution: Ordering: ABDEGFCHI

nodes	pre-visit	post-visit
A	1	12
B	2	11
D	3	6
E	4	5
G	7	10
F	8	9
C	13	18
H	14	17
I	15	16

Tree: AB, BD, DE, BG, GF, CH, HI

Back: ED

Forward: AE, CI

Cross: GD

(b) How many strongly connected components are there?

Solution: $\{D,E\},\{B\},\{G\},\{F\},\{A\},\{C\},\{H\},\{I\}$ are the strongly connected components, so there are 8 strongly connected components.

(c) Let $|E|$ be the number of edges in a simple graph and $|V|$ be the number of vertices. Show that $|E|$ is in $O(|V|^2)$.

Solution:

The maximum number of edges a graph can have is when every vertex is connected to every other vertex. This describes the 'complete' graph and has $\binom{V}{2} \in O(V^2)$ edges.

(d) For each vertex v_i, let d_i be the *degree*- the number of edges incident to it. Show that $\sum d_i$ must be even.

Solution: Each edge in the graph belongs to precisely 2 vertices. Thus, the total number of edges is $\frac{\sum d_i}{2}$. Since there are an integer number of edges, $\sum d_i$ must be even.

(e) An undirected graph G is called *bipartite* if we can separate its vertices into two subsets A and B such that every edge in G must cross between A and B. Show that a graph is bipartite if and only if it has no odd cycles.
Hint: Consider a *spanning tree* of the graph, which is a subset of the graph's edges which allow it to be a tree on all of its vertices.

Solution:

Without loss of generality, assume that the graph is connected. Otherwise, apply the following proof to each connected component.

First, assume a graph has no odd cycles. Do a BFS from some root vertex r. Color a vertex red if it has odd distance from r and black if it has an even distance to r. We now show that the red and black vertices form a bipartition of the graph. Suppose, for the sake of contradiction, that there are two vertices u and v that are both red and are adjacent to each other. Let x be the least common ancestor of u and v in the tree. Since u and v are both red, the xu and xv paths both have even lengths or both have odd lengths. In particular, concatenating the paths shows that there is an even-length path that connects u to v. Adding the uv edge to this path creates an odd cycle, which is a contradiction.

On the other hand, a bipartite graph can never have an odd cycle. Suppose there is an odd cycle. We can try to color the odd cycle. WLOG, assume, the first vertex of the coloring is blue. This uniquely determines the color of the next vertex, and so on. So the odd vertices on this path are blue and the even are red. The last vertex cannot be colored either red or blue, since it is neighbors with the first vertex which is also odd and the previous vertex which is even. Hence, no odd cycle can exist.

(f) A directed acyclic graph G is *semiconnected* if for any vertices A and B there is either a path from A to B or a path from B to A. Show that G is semiconnected if and only if there is a directed path that visits all of the vertices of G.

Solution: First, we show that the existence of a directed path p that visits all vertices implies that G is semiconnected. For any two vertices A and B, just consider the subpath of p. If A appears before B in p, then this subpath will go from A to B. Otherwise, it will go from B to A. In either case, A and B are semiconnected for all pairs of vertices (A,B) in G. This completes this direction.

Now, we show that if the DAG G is semiconnected, then there is a directed path that visits all of the vertices. Consider a topological ordering $v_1, v_2, \ldots, v_n$ of the vertices in G. For any pair of consecutive vertices v_i, v_{i+1}, we know that there is a path from v_i to v_{i+1} or from v_{i+1} to v_i by semiconnectedness. But topological orderings do not have any edges from later vertices to earlier vertices. Therefore, there is a path from v_i to v_{i+1} in G. This path cannot visit any other vertices in G because the path cannot travel from later vertices to earlier vertices in the topological ordering. Therefore, the path from v_i to v_{i+1} must be a single edge from v_i to v_{i+1}. This edge exists for any consecutive pair of vertices in the topological ordering, so there is a path from v_1 to v_n that visits all vertices of G.

Optional redemption file

Submit your *redemption file* for Homework 1 here. If you looked at the solutions and took notes on what you learned or what you got wrong, include them here.