Time of access: 6:10pm

M1.

- Submit on Gradescope within **6 hours of your first access**. Please note down the time of your access now on the top of this page, above the M1 heading.
- Accesses are logged by the system. **Your timeliness is expected.**
- Timestamps in the form of carefully, completely-photographed work sent by email are admissible in a technical emergency.
- Do not share or make the exam accessible to anyone other than yourself. Any suspected violation of this policy will constitute a breach of the University's academic integrity policy and will be reported.
- You can either type your solutions using LaTeX or scan your handwritten work. You will find a LaTeX template in the midterm folder. If you are writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will facilitate the grading.
- **No collaboration and no use of external resources** are allowed for this midterm. Any suspected violation will be investigated and reported. You may consult the textbook, the material and videos linked from the Canvas Modules page and any notes that you took for the course.
- If you absolutely need to contact the teaching team, please use a private message on Ed.

PROBLEM 1 (25 POINTS)

Here is an algorithm that attempts to produce a stable matching between N hospitals and N students.

1. Let M be an arbitrary initial matching between hospitals and students;
2. Let (h,s) and (h',s') be arbitrary pairs in M such that (h,s') form an unstable pair with respect to M, i.e., an unmatched pair such that both parties prefer each other over their current match. If no such pair exists, **halt** with output M.
3. Modify M by swapping the assignments; remove the previous two pairs, and add two new pairs matching (h,s') and (h',s)
4. Goto line 2.

We will show that this algorithm can run forever without producing a stable matching even for $N = 3$. To accomplish this task, denote the hospitals as $H = \{h_1, h_2, h_3\}$, and the students as $S = \{s_1, s_2, s_3\}$. Construct a counterexample by completing the following parts.

(a) Fill in the blanks in the following preference lists for your counterexample. The leftmost entry is the *most* preferred; the rightmost is the *least*.

Solution:

h_1:	s_3,	s_2,	s_1.	s_1:	h_3,	h_2,	h_1
h_2:	s_1,	s_3,	s_2	s_2:	h_3,	h_2,	h_1
h_3:	s_3,	s_1,	s_2	s_3:	h_2,	h_3,	h_1

(b) We assume, without loss of generality, that the initial matching is $\{(h_1,s_1),(h_2,s_2),(h_3,s_3)\}$. Fill in the following table with with the execution of the algorithm that leads to an infinite loop. In the "Iteration #" column, write down the iteration number; in the "Unstable Pair" column, write down the unstable pair that triggers a swap; in the "Matching After Swap" column, write down the matching after the swap is performed.

Solution:

Iteration #	Unstable Pair	Matching After Swap
1	(h_2, s_3)	$(h_1, s_1), (h_2, s_3), (h_3, s_2)$
2	$(h_2, s_1)'$	$(h_1, s_3), (h_2, s_1), (h_3, s_2)$
3	(h_3, s_1)	$(h_1, s_3), (h_2, s_2), (h_3, s_1)$
4	(h_3, s_3)	$(h_1, s_1), (h_2, s_2), (h_3, s_3)$
	$(\_\_, \_\_)$	$(h_1, \_\_), (h_2, \_\_), (h_3, \_\_)$
	$(\_\_, \_\_)$	$(h_1, \_\_), (h_2, \_\_), (h_3, \_\_)$
	$(\_\_, \_\_)$	$(h_1, \_\_), (h_2, \_\_), (h_3, \_\_)$

PROBLEM 2 (25 POINTS)

Give an algorithm that takes as input an undirected, weighted graph $G = (V, E, w)$, with distinct edge weights w, and a subset of edges of $S \subseteq E$, which does not contain a cycle, and returns the spanning tree of G of minimum weight out of all spanning trees containing all edges in S. Prove that your algorithm is correct and give an explicit Big-O bound on its running time as a function of $|E|$ and $|V|$.

Solution: Your solution goes here.

Problem 2:

To create this algorithm we will modify Kruskal's algorithm to account for the subset of edges S.

We will initialize a set K to contain all the edges that are in the subset S. Say this set K has length m while the original subset S has length l.

Code:

```
K ← S
//$w(e_1) < w(e_2) < ... < w(e_m - e_l)$
For i=1 to m-l:
    If $e_i$ not in K:
        If $(K \cup e_i)$ contains no cycles:
            $K \leftarrow (K \cup e_i)$
Return K
```

Correctness: It is impossible for this algorithm to return a cycle as it checks for cycles before adding each edge. We know that the spanning tree returned by this algorithm will contain all the edges in subset S because K was initialized to be S. If we allow E to be the edges returned at any point in this algorithm, at the beginning it E = S. Additionally, at every stage of the algorithm's execution there exists a tree T such that E is a subset of T. If the next edge added to E is in T, then E remains a subset of T, however if it is not in T, then E will necessarily form a cycle. By the definition of our algorithm this is not possible, meaning that the algorithm will run until T manifests.

Time complexity: When Kruskal's algorithm is run normally, it takes slightly longer than this algorithm to execute as K is initialized to the empty set. Thus the complexity of Kruskal's algorithm can be used as an upper bound for this algorithm, meaning it is O(E*log(V)).

PROBLEM 3 (25 POINTS)

Suppose that we have an array $A = [a_0, a_1, a_2, ..., a_{n-1}]$ of integers and an array $S = [s_0, s_1, ..., s_{n-1}]$ of partial sums of A, i.e., each element $s_j = \sum_{i=0}^{j} a_i$ of S is the sum of the first $j+1$ elements of A. For example, we could have that $A = [2, 3, -4, 2, 1, 6, -8, 5, -12, 4]$ and $S = [2, 5, 1, 3, 4, 10, 2, 7, -5, -1]$.

Give a fast algorithm which, provided that the sum of the elements of A is odd, finds an odd element of A. Show that your algorithm is correct and analyze its running time by proving an explicit Big-O bound as a function of n. A solution that runs in $\Omega(n)$ time will receive no credit, as this is trivially achieved by checking elements one-by-one.

Solution: Your solution goes here.

Problem 3:

Algorithm:

If the first element of A is 0 then our job is easy, return 0.
Then we will loop from the start of the array (0) to end (n-1):

```
Last = n-1
First = 0
//This is basically a recursive algorithm, written in a loop
For i = 0 to n-1:
        M = ⌊((Last - First)/2)⌋ //this is a floor function
        If A[M] % 2 == 1: //if midpoint is odd
                Return M
        If (S[Last] - S[mid]) % 2 == 1:
                First = M
        Else If (S[mid] - S[First]) % 2 == 1:
                Last = M
```

Correctness: First we check the first element, if it is odd then our work is done. Then we check the midpoint and return its index if it is odd. Now, it must be true that there is an odd element between elements i and j if S[i] - S[j] is odd. So we test S[Last] - S[mid] to find out if there is an odd element between A[Last] and A[mid]. If there is, we can make the midpoint our new start. If not, then it must be true that S[mid] - S[First] is odd, otherwise the sum of the elements of A would be even. Thus, in this case we can set the endpoint to the midpoint. Then the algorithm is rerun (it is basically recursive) with these new mid and endpoints. Every time this algorithm executes the search list is consolidated such that there is an odd element between the new start and endpoints. Thus the algorithm will reduce the list S until it returns the index of an odd element.

Time complexity: Since we cut the size of the list we are searching in half after each iteration, this algorithm's time complexity is equivalent to a binary search. Thus it is O(log(n)).

PROBLEM 4 (25 POINTS)

Suppose that an emergency vehicle drives from city A to city B over a straight highway that is split in n segments. Each segment i has a length l_i and a speed limit u_i. The driver has special permissions that allow her to violate the speed limit by v units for a total time T. Specifically, for each segment $i \in \{1, 2, \ldots, n\}$, the driver has to choose for which duration $t_i \geq 0$ she will travel with the higher speed $u_i + v$, under the constraint that the total violation time $\sum_{i=1}^{n} t_i$ is at most T.

Devise a polynomial-time greedy algorithm that will allow the driver to go from city A to city B in a minimum amount of time, if she is allowed to violate the speed limit for a total time T and by v units of speed. Prove the correctness of your algorithms and give a Big-O bound on the running time as a function of n.

Hint: Write a closed-form expression for the time needed to traverse a segment i, if the driver violates the speed limit and travels at speed $u_i + v$ for a duration t_i. Notice that t_i cannot be arbitrarily large.

Solution: Your solution goes here.

Problem 4:

To store the data necessary for this problem we will use a two dimensional list H (set containing sets with 3 elements). For each segment i, H(i) = {i, u_i, l_i}. Now we need to preprocess our list, which we will do by sorting it from smallest to largest using the following rule:

Syntax: H(i)[0] = i, H(i)[1] = u_i, H(i)[2] = l_i
We say that H(j) > H(i) if: (H(j)[1] > H(i)[1]) or (Hj)[1] = H(i)[1] and H(j)[2] >H(i)[2])
We say that H(j) = H(i) if: H(j)[1] = H(i)[1] and H(j)[2] = H(i)[2]

After preprocessing, we will use the following greedy algorithm:

```
Time = 0
While Time < T and i < n:
        Time = Time + ( H(i)[2]/ (H(i)[1] + $v$) )
        i = i + 1
Return  i, Time
```

All the values of i less than the variable i outputted by this algorithm are segments for which we can speed. Thus, for all i < n, we speed for distance l_i if H(i)[0] < i and we do not speed if H(i)[0] > i.

Correctness: This algorithm works because we achieve the minimum amount of time by speeding when the speed limit is lowest. We can compare our solution to the optimal solution. Consider our solution, the sorted list H, found using the rule above. Since H is ordered from the smallest *u* to the largest, it will maximize the time saved by speeding over each section of road of a certain standard length. Thus the time it takes to travel the road over a certain standard length with my algorithm must be at least as good as the optimal solution (and by extension the time it takes to travel the whole road using my algorithm must be at least as fast as the optimal solution).

Time complexity: The preprocessing stage can be done in polynomial time; it is at most O(n^2). The second stage of the algorithm will loop for at most *n* iterations. Thus an upper bound for this algorithm is O(n^2).