homework 2

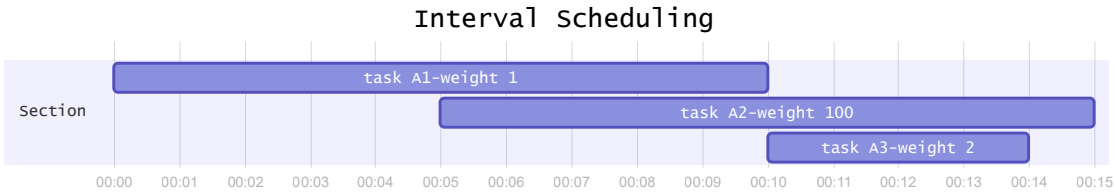
Name: Yifan Zhang

University ID: yz113

Problem 1

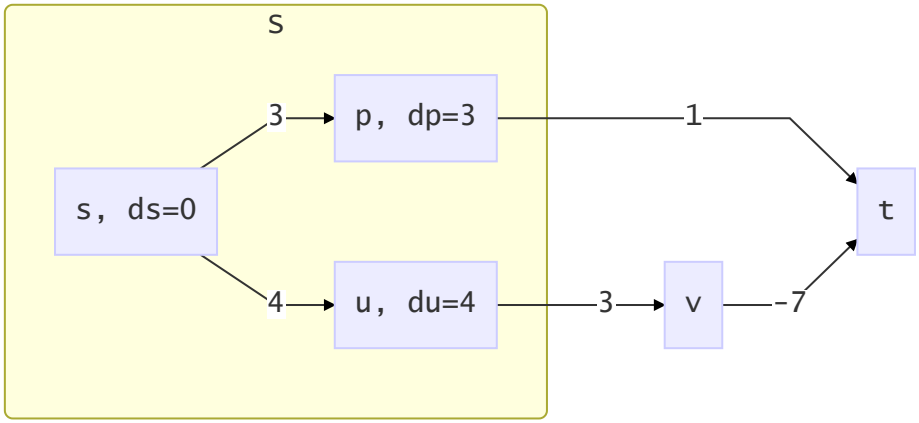
> 1. In Interval Scheduling problem, if each request not only has a starting time and a finishing time but also has weight, and we want to maximize the total weights of intervals we selected. Can we still use the greedy algorithm that we learned during the lecture (process according to the finishing time)? If yes give a proof; if no give an counterexample.
> 2. Give a counter example showing that Dijkstra's algorithm does not work for graphs with negative edges. Please simulate the running of Dijkstra's algorithm on your counter example and explain why Dijkstra's algorithm fails in this case.

1. the answer is no. Here is a counterexample:



   If we still using a greedy algorithm that process according to the finishing time, task A1 and task A2 will be execute and task A3 will be dropped. However A3's weight is much more than A1's weight plus A2's weight.

2. This is the counter example showing that Dijkstra's algorithm does not work for graphs with negative edges:



In next step, the algorithm will add node t in to S and define d(t) = 4, because d(t) = l(p, t) + d(p) = 4.

However this is wrong, because if we go from s-->u-->v-->t, we will have a smaller d(t) = 0. As a result, Dijkstra's algorithm does not work for graphs with negative edges.

Problem 2

> During the lecture, we learned a greedy algorithm to solve the Scheduling All Intervals problem. We also learned that a naive implementation of the greedy algorithm would take $O(n^2)$ time. Please describe a more efficient implementation of the greedy algorithm with $O(nlogn)$ running time?

First, our algorithm sorts the intervals according to the start times.

Then we can use a min heap to manage the end time of the interval running on each processor. At every time when we need to launch a new interval(in the order of the intervals start time), we check the root node of our min heap for the earliest idle processor. If the earliest intervals end time recorded in root node is later than the incoming interval, we will assigns the interval to a newly added processor, and add a new node to our min-heap to record the newly added processor's idle time. It will take $O(log\ n\ )$ to insert a node to the min heap. If the earliest interval end time is earlier than the incoming interval's start time, we will use this processor and update this min-heap node's value(the root node) to the new interval's end time and sort our min heap. It will take $O(log\ n\ )$ to sort a changed root node in min heap.

Our new algorithm will need $O(log\ n\ )$ on every step, and there are n steps. Overall, its running time is $O(nlog\ n\ )$.

Problem 3

> On Thanksgiving day, you arrive on an island with n turkeys. You've already had thanksgiving dinner, so you don't want to eat the turkeys, but you do want to wish them all a Happy Thanksgiving.
>
> However, the turkeys each have very different sleep schedules. Turkey i is awake only in a single closed interval $[a_i, b_i]$. Your plan is to stand in the center of the island and say loudly "Happy Thanksgiving!" at certain times $t_1, \ldots, t_m$. Any turkey who is awake at one of the times t_j will hear the message. It's okay if a turkey hears the message more than once, but

> you want to be sure that every turkey hears the message at least once.
>
> Design a greedy algorithm which takes as input the list of intervals $[a_i, b_i]$ and outputs a list of times $t_1, \ldots, t_m$ so that m is as small as possible and so that every turkey hears the message at least once. Your algorithm should run in time $O(nlog\ n\ )$. Prove that your algorithm is correct.

- Algorithm description

  1. Sort every turkey's wake up time as a list l. It takes $O(nlog\ n\ )$.
  2. Use a min heap h to manage turkey's sleep time.
  3. Pick up the earliest wake up turkey e in l, if turkey e's wake up time is later than the earliest sleep time t marked in the root node of min heap h, schedule a "Happy Thanksgiving!" at time t, and free the min heap. This will take $O(1)$. Else, add l's sleep time in to the min heap h. It takes $O(log\ n\ )$ to sort the min heap.
  4. If the list l is not empty, go back to step 3.
  5. When the list l is empty and min heap h is not empty, schedule a "Happy Thanksgiving!" at the sleep time recorded in root node(which is the earliest sleep time in current heap).

  Over all, this algorithm's time complexity is $O(nlog\ n\ )$.

- Correctness proof

  1. Every turkey hears the message at least once

     First, we know that every turkey has been pushed in to a list ordered by their wake up time. Besides we know that every turkey will be popped out and make a comparation with the root node of min heap h. Only if the next turkey's wake up time t' is later than the earliest sleep time t in current min heap h, we will schedule a "Happy Thanksgiving!" at time t, and delete the min heap, and add the turkey with wake up time t' in to a new empty min heap. It means that every turkey will be added into a min heap.

     We only need to prove that every turkey that deleted from min heap h will hear the message at least once.

     We add next turkey n into min heap h only if n's wake up time is not later than the earliest sleep time in current heap, which means that at the time t when we schedule a "Happy Thanksgiving!" , every turkey in the heap will be able to hear this message because all of their sleep time is later than t. Meanwhile, every turkey has already woke up because the list from which we choosing turkey is sorted by their wake up time. As a result, every turkey in min heap will be able to hear the message.

     As a result, every turkey will be able to hear the message.

  2. m is as small as possible

     For a time slice $[t_i, t_j]$, t_i is a arbitrary turkey's wake up or sleep time and t_j is the nearest following any other turkey's wake up or sleep time. Namely, in $[t_i, t_j]$, there is no turkey wake up or fall asleep. If we say "Happy Thanksgiving!" at any time in this time slice, there will not be more or less turkey can hear the message. We name two holiday congratulations are **equivalent** when they are in such a time slice.

     Suppose that our algorithm will end with m times holiday congratulations. Suppose there is another schedule O that has less holiday congratulations time n $(n < m)$ and can make sure every turkey can hear the holiday congratulations too. Our algorithm's output is $t_1, \ldots, t_m$ and O's output is $t'_1, \ldots, t'_m$.

So there should be exist a q that from $q\ th$ on,t_q and t'_q are not **equivalent**. However, t'_q should be earlier than t_q, because if , t'_q is later than t_q, O will miss a turkey. This is because our algorithm schedule a holiday congratulation at current turkey's earliest sleep time. If O schedule a holiday congratulation later than t_q, O will lost this earliest slept turkey. As a result, t'_q should be earlier than t_q. This means that up to $q\ th$ holiday congratulation, O has not make holiday congratulation to more turkey than our algorithm, because t'_q is earlier than t_q.

For $q+1\ th$ holiday congratulation, the story will not change, for the same reason. If O schedule a holiday congratulation later than t_{q+1}, O will lost earliest slept turkey in this round, because O did not sent holiday congratulation to more turkey than our algorithm, so this earliest slept turkey will not be messaged at t'_q. This means that from t'_q on, O's every time schedule will earlier than our algorithm. If O has less holiday congratulations times than our algorithm, O will definitely make some turkey(s) who wake up late can't receive the message.

As a result, O does not exist. So in our algorithm, m is the smallest.

Problem 4

> Consider the following two statements:
>
> 1. The first *k* edges chosen by Kruskal's algorithm have the following property: there is no cheaper acyclic subgraph of G with *k* edges. (Assume edge costs are distinct)
> 2. The first *k* edges chosen by Prim's algorithm (starting from some "root node" *r*) have the following property: there is no cheaper connected subgraph of *G* containing the node *r* along with *k* other nodes. (Assume edge costs are distinct)
>
> Give a proof if the statement is true or a counterexample if the statement is false.

1.

Proof:

Suppose there exist a cheaper acyclic subgraph s of G with k edges. As a result there must exist a first (sort according to the edge's weight) edge q and the $q\ th$ edge's weight in s is less than the $q\ th$ edge's weight chosen by Kruskal's algorithm.

However according to Kruskal's algorithm, at the $q\ th$ edge choice we will choose the lightest edge. The only reason we don't choose the lightest edge is because if we add this edge into our tree, there will be a cycle. As we known, s is a acyclic subgraph.

As a result, q does not exist, which means that s is not exist. Namely, there is no cheaper acyclic subgraph of G with *k* edges.

2.

Proof:

Suppose there exist a cheaper connected subgraph s of G with k edges. As a result there must exist a first (sort according to the edge's weight) edge q and the $q\ th$ edge's weight in s is less than the $q\ th$ edge's weight chosen by Prim's algorithm.

There are 2 cases for this $q\ th$ edge:

1. This edge will create a cycle.

   In this case, we can just ignore this $q\ th$ edge and take look in following edges, because the subgraph s must contain node r with *k* other nodes. If there is a edge will create a cycle in s, that means s will have one more edge than Prim's tree. If the rest edges' weight is heavier than or equal with Prim's tree s will not be cheaper than Prim's tree.

2. This edge will not create a cycle.

   According to Prim's algorithm, at the $q\ th$ edge choice we will choose the lightest edge. The only reason we don't choose the lightest edge is because if we add this edge into our tree, there will be a cycle. However, in this case, this edge will not create a cycle.

As a result, q does not exist, which means that s is not exist. Namely, there is no cheaper connected subgraph of G containing the node r along with *k* other nodes.

Problem 5

> Let G be a connected weighted undirected graph. In class, we defined a minimum spanning tree of *G* as a spanning tree *T* of *G* which minimizes the quantity
>
> $$X = \sum_{e \in T} w_e$$
>
> where the sum is over all the edges in T, and w_e is the weight of edge e.
>
> Define a "minimum-maximum spanning tree" to be a spanning tree that minimizes the quantity
>
> $$Y = \max_{e \in T} w_e$$
>
> That is, a minimum-maximum spanning tree has the smallest maximum edge weight out of all possible spanning trees.
>
> 1. Prove that a minimum spanning tree in a connected weighted undirected graph G is always a minimum-maximum spanning tree for G.
> 2. Show that the converse to part (1) is not true. That is, a minimum-maximum spanning tree is not necessarily a minimum spanning tree.

1.

   Proof:

   Suppose that in G there exist a minimum spanning tree mst, and a minimum-maximum spanning tree $mmst$, and $mmst$'s maximum edge weight mmw_e is larger than mst's maximum edge weight mw_e.

   mw_e connects two parts of the mst. However, there must exist another edge with less weight that can connect these two parts too, because $mmst$ can connect the whole spanning tree without using any edge whose weight is equal with or larger than mw_e. In this case, our mst is not a minimum spanning tree because it does not satisfy the **Cut property** we proved in class.

   It is proved by contradiction.

2.

   First we generate a Minimum Spanning Tree mst in G. From part(1) we know that mst is also a minimum-maximum spanning tree. Then we arbitrarily cut mst in to two trees, there should be some edges that connect these two parts. Now we delete the smallest edge and add a new edge e from G into our tree, and the new edge e only need to satisfy that: firstly, e is not the smallest edge between these two parts; secondly, e's weight is not larger than the maximum edge weight. If we can't find a e in this cut, we will restore the deleted edge and do another cut. If there does not exist an edge e from G at any cut of mst, that means that G itself is a minimum spanning tree, and we don't consider this specific situation. Finally, we have a new spanning tree mst'. However, mst' is not a minimum spanning tree anymore, but it is still a minimum-maximum spanning tree.

As a result, a minimum-maximum spanning tree is not necessarily a minimum spanning tree.

Problem 6

> One of the first things you learn in calculus is how to minimize a differentiable function such as $y = ax^2 + bx + c$, where $a > 0$. The Minimum Spanning Tree Problem, on the other hand, is a minimization problem of a very different flavor. One can ask what happens when these two minimization issues are brought together, and the following question is an example of this.
>
> Suppose we have a connected graph $G = (V, E)$. Each edge e now has a time varying edge cost given by a function $f_e : R \to R$. Thus, at time t, it has cost $f_e(t)$. We will assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t; we will denote this function $c_G(t)$. A natural problem then becomes: find a value of t at which $c_G(t)$ minimized.
>
> Suppose each function f_e is a polynomial of degree 2: $f_e(t) = a_e t^2 + b_e t + c_e$, where $a_e > 0$. Give an algorithm that takes the graph G and the values $(a_e, b_e, c_e) : e \in E$ and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G. You may assume that arithmetic operations on the numbers (a_e, b_e, c_e) can be done in constant time per operation.

First we can know that MST will change only if the size relationship between the edges changes. Besides we know that the relationship between the edges will only change after the edges' functions intersection.

Algorithm description:

1. We compute every intersection between edges' functions. At most this will cost $O(n^2)$.
2. We compute every possible MST's minimum cost between every adjacent intersection. Every MST's cost will be a function of time: $f_{mst}(t) = at^2 + bt + c$, and $a = \sum^{e \in E} a_e; b = \sum^{e \in E} b_e; c = \sum^{e \in E} c_e$. According to assumption in this problem, it will take $O(1)$ to find a t for a minimum $f_{mst}(t)$ in every interval and we have at most $n * (n-1) - 1$ intervals. So this step will cost $O(n^2)$.
3. Find the minimum MST's cost and return the corresponding t. This will take $O(n)$.

Overall, our algorithm's time complexity is $O(n^2)$.

Problem 7

> We are given two arrays of n points on the number line: red points $r_1, r_2, \ldots, r_n \in R$ and blue points $b_1, b_2, \ldots, b_n \in R$. You may assume that all red points are distinct and all blue points are distinct.
>
> We want to pair up red and blue points in the following way: find a bijection $\pi : 1, \ldots, n \to 1, \ldots, n$ so that r_i is matched with $b_{\pi(i)}$. Note that each red point is matched with a unique blue point and vice versa. We would like to minimize the following objective function:
>
> $$\sum_{i=1}^{n} |r_i - b_{\pi(i)}|$$

> Design an $O(nlog\ n\ )$ time algorithm that finds the matching (bijection) that minimizes the sum of distances between the matched points. Prove the correctness of the algorithm and analyze its running time.

Sort elements in r and b according their value and we have arrays r' and b'. For $i\ th$ element in r', it will be matched to $i\ th$ element in b'. Its time complexity is $O(nlog\ n)$.

Correctness Proof:

If we pair $r'[i]$ with $b'[j]$, and $b'[i]$ with $r'[j]$. and $r'[i] > r'[j], b'[i] > b'[j]$

1. $r'[i] > r'[j] > b'[i] > b'[j]$, which is the same as $b'[i] > b'[j] > r'[i] > r'[j]$

   In this case, $|r'[i] - b'[j]| + |r'[j] - b'[i]| == |r'[i] - b'[i]| + |r'[j] - b'[j]|$. In this case, there is no difference between our algorithm's pair and this pair.

2. $r'[i] > b'[i] > r'[j] > b'[j]$, which is the same as $b'[i] > r'[i] > b'[j] > r'[j]$

   In this case, $|r'[i] - b'[j]| + |r'[j] - b'[i]| > |r'[i] - b'[i]| + |r'[j] - b'[j]|$. In this case, our algorithm's pair is better than this pair.

3. $r'[i] > b'[i] > b'[j] > r'[j]$, which is the same as $b'[i] > r'[i] > r'[j] > b'[j]$

   In this case, $|r'[i] - b'[j]| + |r'[j] - b'[i]| > |r'[i] - b'[i]| + |r'[j] - b'[j]|$. In this case, our algorithm's pair is better than this pair.

As a result, if we change any two pairs in out algorithm's output, we will never get a better pair. Besides, by switching pairs multiple times, we can end with any bijection. As a result, there will not exist a bijection that has less of the sum of distances than our algorithm's output. Namely, our algorithm's output can minimizes the sum of distances between the matched points.

Problem 8

> Jane loves tomatoes! She eats one tomato every day, because she is obsessed with the health benefits of the potent antioxidant lycopene and because she just happens to like them very much.
>
> The price of tomatoes rises and falls during the year, and when the price of tomatoes is low, Jane would naturally like to buy as many tomatoes as she can. Because tomatoes have a shelf-life of only d days, however, she must eat a tomato bought on day i on some day j in the range $i \leq j < i + d$, or else the tomato will spoil and be wasted. Thus, although Jane can buy as many tomatoes as she wants on any given day, because she consumes only one tomato per day, she must be circumspect about purchasing too many, even if the price is low.
>
> Jane's obsession has led her to worry about whether she is spending too much money on tomatoes. She has obtained historical pricing data for n days, and she knows how much she actually spent on those days. The historical data consists of an array $C[1, \ldots, n]$, where $C[i]$ is the price of a tomato on day i. She would like to analyze the historical data to determine what is the minimum amount she could possibly have spent in order to satisfy her tomato-a-day habit, and then she will compare that value to what she actually spent.
>
> Give an $O(n)$ time algorithm to determine the optimal offline purchasing strategy on the historical data. Given d, n , and $C[1, \ldots, n]$, your algorithm should output $B[1, \ldots, n]$, where $B[i]$ is the number of tomatoes to buy on day i. Please also prove the correctness of your algorithm. You may get partial credits with slower algorithm.

- Algorithm description

```
index := n;
B[n] := [0,....0]
while index >= 1:
    store_date := 0;
    minimum_index := index;
    while store_date <= d:
        if C[index - store_date] < C[index]:
            minimum_index := index - store_date;
        if index - store_date == 1:
            break;
        store_sate := store_date - 1;
    C[minimum_index] := C[minimum_index] + 1;
    index := index - 1;
return B;
```

Time complexity is $O(dn)$, but d is a constant, so the time complexity is $O(n)$.

- Correctness

  What our algorithm do is to find the cheapest tomato's day in $[i-d, i]$ for everyday i, and add 1 to the cheapest day's tomato purchase schedule for day i. Namely, everyday, Jane will eat the cheapest tomato from d days ago to today, which means that the tomato she eats every day are the cheapest of all possible. The correctness is obvious.