CPSC 320 2021W2: Assignment 3

This assignment is due **Monday October 31, 2022 at 22:00 Pacific Time**. Assignments submitted within 24 hours after the deadline will be accepted, but a penalty of up to 15% will be applied. Please follow these guidelines:

- Prepare your solution using LaTeX, and submit a pdf file. Easiest will be to use the .tex file provided. For questions where you need to select a circle, you can simply change `\fillinMCmath` to `\fillinMCmathsoln` .
- Enclose each paragraph of your solution with `\begin{soln}Your solution here...\end{soln}`. Your solution will then appear in dark blue, making it a lot easier for TAs to find what you wrote.
- Start each problem on a new page, using the same numbering and ordering as this assignment handout.
- Submit the assignment via GradeScope at `https://gradescope.ca`. Your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**.
- After uploading to Gradescope, link each question with the page of your pdf containing your solution. There are instructions for doing this on the CPSC 121 website, see `https://www.students.cs.ubc.ca/~cs-121/2020W2/index.php?page=assignments&menu=1&submenu=3`. Ignore the statement about group size that is on that page.

Before we begin, a few notes on pseudocode throughout CPSC 320: Your pseudocode should communicate your algorithm clearly, concisely, correctly, and without irrelevant detail. Reasonable use of plain English is fine in such pseudocode. You should envision your audience as a capable CPSC 320 student unfamiliar with the problem you are solving. If you choose to use actual code, note that you may **neither** include what we consider to be irrelevant detail **nor** assume that we understand the particular language you chose. (So, for example, do not write `#include <iostream>` at the start of your pseudocode, and avoid language-specific notation like C/C++/Java's ternary (question-mark-colon) operator.)

Remember also to **justify/explain your answers**. We understand that gauging how much justification to provide can be tricky. Inevitably, judgment is applied by both student and grader as to how much is enough, or too much, and it's frustrating for all concerned when judgments don't align. Justifications/explanations need not be long or formal, but should be clear and specific (referring back to lines of pseudocode, for example). Proofs should be a bit more formal.

On the plus side, if you choose an incorrect answer when selecting an option but your reasoning shows partial understanding, you might get more marks than if no justification is provided. And the effort you expend in writing down the justification will hopefully help you gain deeper understanding and may well help you converge to the right selection :).

Ok, time to get started...

1 Statement on Collaboration and Use of Resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments in CPSC 320 (see `https://www.students.cs.ubc.ca/~cs-320/2022W1/index.php?page=assignments&menu=1&submenu=3`).

   ◯ Yes ◯ No

2. We used the following resources (list books, online sources, etc. that you consulted):

3. One or more of us consulted with course staff during office hours.

   ◯ Yes ◯ No

4. If one or more of us collaborated with other CPSC 320 students: none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ◯ Yes ◯ No

   If yes, please list their name(s) here:

5. If one or more of us collaborated with or consulted others outside of CPSC 320: none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ◯ Yes ◯ No

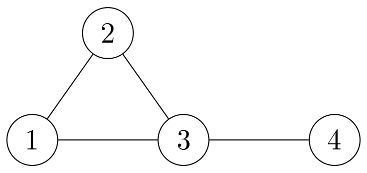
   If yes, please list their name(s) here:

2 Colour Me Puzzled (Grid-Free Version)

We are given a graph $G = (V, E)$. We want to colour each vertex with one of k colours so that two end points of any edge in E receive different colours. This is called a *vertex k-colouring* of the graph.

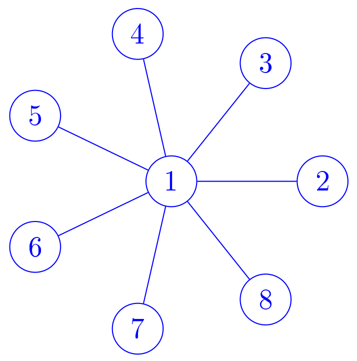
Recall that the *degree* of a vertex $v \in V$ is the number of edges incident on v. Let d be the *maximum degree* in the graph.

Example:



The maximum degree d in this graph is 3. The graph has a 3-colouring (by using different colours for nodes 1, 2, and 3 and colouring node 4 the same as 1 or 2) and a 4-colouring (by using a different colour for every node), but does not have a 2-colouring.

1. **[1 mark]** For any value of d, describe a graph with the maximum degree d that can be coloured by two colours. A star graph has one central vertex that connected by an edge to every other vertex. There are no other edges. Hence, a star graph with n vertices will have one vertex with degree $n-1$ and $n-1$ vertices with degree one. The maximum degree $d = n-1$, yet, it can be coloured with two colours: use one colour for the central vertex and the other colour for all other vertices.



Any bipartite graph (remember these from take-home test 1?) with maximum degree d would also work as a solution: we could use one colour for the nodes on the left and one for the nodes on the right.

2. **[2 marks]** Design a greedy algorithm that will colour vertices of G with at most $d + 1$ colors. Your algorithm should consider vertices in an arbitrary order and then greedily choose the colour of each vertex.

   Order vertices arbitrarily. Colour the first vertex with colour 1. Then choose the next vertex v and colour it with the lowest-numbered colour that has not been used on any previously-coloured vertices adjacent to v. If all previously-used colours appear on vertices adjacent to v, this means we must introduce a new colour and number it.

3. **[2 marks]** Explain why your algorithm works correctly (i.e., why it will use at most $d + 1$ colours).

   Consider the step of assigning a colour to a vertex v. Because each v has at most d neighbours, the worst case is that v has d neighbours which have each already been assigned a different colour, and we will need to introduce a colour $d+1$. But if we use $d+1$ colours, we're guaranteed to always have a colour available for every new vertex (because a new vertex can't have more than d neighbours that have already been coloured).

   Important note here: we have claimed here that our algorithm has a certain "good" property (namely, that it uses at most $d+1$ colours, as desired), but we **are not** claiming that the algorithm is **optimal**, in the sense of using the fewest possible colours. It is overwhelmingly likely (but perhaps not 100% certain) that the problem of colouring a graph with the fewest possible colours cannot be optimally solved by any greedy algorithm. We'll learn more about this in our unit on NP-completeness!

4. **[4 marks]** Now, assume that there is at least one vertex v in V with degree **less than** d. Design a greedy algorithm that will colour vertices of G with at most d colours. You should proceed by first **ordering the vertices** in some way, and then assigning colours using the greedy strategy you developed previously. You should explain why your algorithm uses no more than d colours.

   We want order vertices in such a way that every vertex has at most $d - 1$ neighbors among the preceding, already-coloured vertices. This will be true for a vertex that has at least one neighbor among the following vertices or if the vertex is the special vertex v that has degree less than d (we are guaranteed to have one like that). It's a good idea to put v at the end of the node ordering, since we know that, if we use d colours, we can safely colour v last and still have a colour available for it. Now, we want an ordering so that each vertex except the last has a "forward" edge (edge going from this vertex to some neighbor further down the list). How does a graph that contain only these forward edges look like? Well, it looks like a tree rooted at the last vertex. So, to find such order, let's first find a tree rooted at v and then order vertices so that "up-tree" edge is a forward edge in this order.

   One possible procedure: Build a BFS tree from v with all edges pointing towards the parent. Note that v is the root in this tree. Find a *topological ordering* of this tree (an alternative way to think about this if you aren't comfortable/familiar with topological orderings: start with vertices at the deepest level of the BFS tree, and add nodes as you go up the levels of the tree, considering the nodes within each level in any order). Repeat the colouring algorithm from question 2, using this ordering of the nodes. Each vertex but the last one has an adjacent vertex to the right (i.e., later) in the list. Therefore, each vertex has at most $d-1$ vertices *earlier* in the list (and this is also true of the last), which means we will need at most d colours.

3 Shelter Skelter

You're the manager of an animal shelter, which is run by a few full-time staff members and a group of n volunteers. Each of the volunteers is scheduled to work one shift during the week. There are different jobs associated with these shifts (such as caring for the animals, interacting with visitors to the shelter, handling administrative tasks, etc.), but each shift is a single contiguous interval of time. Shifts cannot span more than one week (e.g., we cannot have a shift from 10 PM Saturday to 6 AM Sunday). There can be multiple shifts going on at once.

You'd like to arrange a weekly meeting with your staff and volunteers, but you have too many volunteers to be able to find a meeting time that works for everyone. Instead, you'd like to identify a suitable subset of volunteers to instead attend the weekly meeting. You'd like for every volunteer to have a shift that overlaps (at least partially) with a volunteer who is attending the meeting. Your thinking is that you can't personally meet with every single volunteer, but you would like to at least meet with people who have been working with every volunteer (and may be able to let you know if a volunteer is disgruntled or having any difficulties with their performance, etc.). Because your volunteers are busy people, you want to accomplish this with the fewest possible volunteers.

Your volunteer shifts are given as an input list V, and each volunteer shift v_i is defined by a start and finish time (s_i, f_i). Your goal is to find the smallest subset of volunteer shifts such that every shift in V overlaps with at least one of the chosen shifts. A shift $v_i = (1,4)$ overlaps with the shift $v_j = (3,5)$ but not with the shift $v_k = (4,6)$.

1. **[4 marks]** Give an efficient algorithm to solve this problem.

   First, a simple statement of the algorithm without worrying about efficiency:

   > Mark all shifts as uncovered. While not all shifts are covered, consider the uncovered shift v' with the earliest end time. Of the shifts that intersect with v', add to the solution the shift v with the latest end time. Mark as covered all shifts that intersect with v.

   Now, to think about how to implement this efficiently. A straightforward implementation of the algorithm above would run in something like $O(n^2)$ time. To improve this, we need to be able to avoid checking all shifts at each step of the solution (or something like all shifts before v/v' or all shifts after v/v' in some sorted order, as this will also give us an $O(n^2)$ time complexity). As a start, we can sort all shifts by start time. This will allow us to check all later-starting shifts that overlap with a particular shift v without necessarily needing to iterate through the entire list (unless v actually overlaps with every shift), because we can stop once we reach a shift whose start time is later than the end time of v.

   To avoid ever needing to check all the way to the beginning of the list for overlapping shifts, an obvious (but incorrect) solution is to remove all shifts that are covered by v at each step. But, sometimes a previously covered shift **does** need to be part of the solution – as in, e.g., $V = [(1,2),(1,6),(2,3),(4,12),(6,7),(8,9)]$, where the first step of the algorithm will select $v = (1,6)$, but then the correct choice for the next interval is the (already covered) shift $(4,12)$. What we **can** do is delete (or otherwise mark as "do not consider as possible choices for subsequent shifts") all shifts that end at or before the end of $v = (1,6)$ (because we know these can't be part of the solution), and mark as covered (but don't remove from consideration) the shifts that are covered by v but end later.

In the pseudocode below, we keep track of uncovered shifts in an array called `U`, and the shifts that could still be part of the solution (which includes all uncovered shifts, in addition to all covered shifts that end later than the previously selected v) in an array called `S`.

```
Algorithm FindShifts(V):
   Sort V by increasing start time
   U ← V   % uncovered shifts
   S ← V   % shifts that could be part of the solution
   Soln ← []

   While U is not empty:
      % assume 1-based indexing
      Let EarlyU ← the set of shifts in U with s < U[1].f
      Let v' ← the shift in EarlyU with the earliest finish time
      Let SBeforeVp ← the set of shifts in S with s < v'.f

      Let v ← the shift in SBeforeVp with the latest finish time
      Add v to Soln
      Let UCoveredByV ← the set of shifts in U with s < v.f
      Remove from U all shifts in UCoveredByV
      SBeforeV ← the set of shifts in S with s < v.f
      Let NoLongerInS ← the set of shifts in SBeforeV with f ≤ v.f
      Remove from S all shifts in NoLongerInS

  return Soln
```

2. **[2 marks]** Briefly justify a good asymptotic bound on the runtime of your algorithm.

Sorting V by increasing start time takes $O(n \log n)$ time. The while loop analysis is trickier, but let's consider how many times a particular shift can be considered in the iterations through U and S. Let's start with U. The elements of U we iterate through at each step of the while loop are those in `EarlyU` and those in `UCoveredByV`. However, `EarlyU` is clearly a subset of `UCoveredByV` (because v could always be $U[1]$, and if it isn't it must mean that v ends later), and everything in `UCoveredByV` is removed at the end of the loop. Thus, no element of U will be accessed in more than two loop iterations (we will need to access the first element *after* `UCoveredByV` to make sure we have all the elements in that set), and all work within the loop takes at most linear time in the number of elements accessed (assuming that each array deletion takes constant time), so the iterations through U take linear time total.

With S, the case is less straightforward because not all elements of S that we access in forming `SBeforeVp` or `SBeforeV` will be deleted at the end of the loop. The set `SBeforeVp` is a subset of `SBeforeV`, because the end time of v is later than or equal to that of v' (because v could be equal to v', so if it isn't it must mean v ends later). There may be some elements of `SBeforeV` (those with $f > v.f$) that are not deleted at the end of the loop iteration. However, these will need to be deleted by the end of the **next** loop iteration (unless all shifts are covered, in which case the algorithm will simply terminate): a non-deleted element of `SBeforeV` needs to start before the next v' (because the next v' has to start after the current $v.f$, and in order for an element to be in `SBeforeV` it must start before $v.f$). This means that, at the next iteration, a non-deleted element of `SBeforeV` either becomes the next selected v, or it ends earlier than the next selected v – in either case, at the next iteration it will be in the set `NoLongerInS`, and will therefore be deleted from S. Thus, no element of S will be accessed in more than three loop iterations (again, at each step we will need to access the first element after `SBeforeV`), and all work within the loop takes at most linear time in the number of elements accessed, meaning the iterations through S will also take linear time total.

Putting all this together, the sorting takes $O(n \log n)$ and the while loop takes $O(n)$, so the total running time of the algorithm is $O(n \log n)$. Note that, because we've only awarded 2 points to this question, you can earn full credit here without your explanation being quite so thorough as ours (additionally, if you came up with a different algorithm, the runtime analysis may have been quite straightforward).

A naïve implementation of the "simple statement of the algorithm" we gave in question 1 would run in something like $\Theta(n^2)$ time. Without pre-sorting by start time, we would need to iterate through the entire list to find all shifts that intersect with a given one, which will lead to an overall $\Theta(n^2)$ runtime. Additionally, if you didn't come up with some way to remove earlier shifts from consideration in choosing the next shift, you would also get a $\Theta(n^2)$ runtime (because you would potentially need to search all earlier-starting shifts to correctly find the one with the latest end time). If your answer to 4.1 uses any kind of repeated "intersects with" calculation **without** first appropriately sorting the array **and** removing earlier-sorted elements from consideration at each step, we will likely consider it to be a $\Theta(n^2)$ (and therefore not optimally efficient) algorithm.

3. **[6 marks]** Prove the correctness of your algorithm – that is, prove that your set of selected shifts will (a) overlap with all volunteer shifts in V, and (b) contain as few shifts as possible. You should do this by identifying some measure under which your solutions "stays ahead" of all other solutions.

Let V' denote the list of shifts sorted by increasing start time. Let $\mathcal{G} = [v_1^G, v_2^G, \ldots, v_k^G]$ denote the shifts selected by the greedy algorithm, and let $\mathcal{O} = [v_1^O, v_2^O, \ldots v_j^O]$ denote an optimal (i.e., smallest possible) selection of shifts. Assume that the selected shifts in both $\mathcal{G}$ and $\mathcal{O}$ appear in the same order as in V' (i.e., sorted by increasing start time).

Let $L_i(\mathcal{G})$ and $L_i(\mathcal{O})$ denote the index within V' (the sorted list of shifts) of the **last shift covered by the first i shifts** of $\mathcal{G}$ and $\mathcal{O}$, respectively. We now prove by induction that, for any $1 \leq i \leq k$, $L_i(\mathcal{G}) \geq L_i(\mathcal{O})$, and there are no shifts starting before the shift at position $L_i(\mathcal{G})$ that are not covered by the selected shifts $v_1^G, v_2^G, \ldots v_i^G$. In this sense, the greedy algorithm "stays ahead" of the optimal solution with each selected shift by covering at least as much of V', while not leaving any earlier shifts uncovered.

For the first shift ($i = 1$), our greedy algorithm chooses the shift with the latest end time of all the shifts that overlap with the earliest-ending shift. Thus, $L_1(\mathcal{G}) \geq L_1(\mathcal{O})$: if $L_1(\mathcal{O})$ were greater than $L_1(\mathcal{G})$, it would mean the first shift selected by the optimal solution ended later than the first shift selected by the greedy solution. This would necessarily mean that the earliest-ending shift of V' was uncovered (because if it were possible to select a first shift with a later end time while still covering the earliest-ending shift, the greedy algorithm would have selected that as the first shift), which would mean that $\mathcal{O}$ was invalid (and therefore not optimal). Additionally, there are no uncovered shifts starting before $L_1(\mathcal{G})$, because there is no shift that ends before the earliest-ending shift (which we know to have been covered).

Now, consider an arbitrary integer i where $2 \leq i \leq k$, and assume that (1) $L_{i-1}(\mathcal{G}) \geq L_{i-1}(\mathcal{O})$, and (2) the greedy algorithm covers all shifts in V' occurring before index $L_{i-1}(\mathcal{G})$. To have $L_i(\mathcal{O}) > L_i(\mathcal{G})$ we would need to have the end time of v_i^O (the ith shift of the optimal solution) be later than the end time of v_i^G (the ith shift of the greedy solution). But this is impossible, because the greedy algorithm will have chosen the latest-ending shift among all shifts that cover the earliest-ending shift after index $L_{i-1}(\mathcal{G})$, and the optimal solution also needs to cover this shift – because, by the inductive hypothesis (1) that $L_{i-1}(\mathcal{G}) \geq L_{i-1}(\mathcal{O})$, this shift is not already covered by the first $i-1$ shifts in $\mathcal{O}$. Again, we arrive at the conclusion that this would mean that $\mathcal{O}$ is invalid (and therefore not optimal), because it leaves uncovered the earliest-ending shift after index $L_{i-1}(\mathcal{G})$ (and possibly earlier shifts as well, if $L_{i-1}(\mathcal{G})$ is strictly greater than $L_{i-1}(\mathcal{O})$). We can also conclude that the greedy algorithm covers all shifts in V' before index $L_i(\mathcal{G})$: by the inductive hypothesis (2) it covers all shifts before index $L_{i-1}(\mathcal{G})$, and because it chooses the latest-ending shift that covers the earliest-ending shift after $L_{i-1}(\mathcal{G})$, it covers all shifts up to the one at index $L_i(\mathcal{G})$.

The greedy algorithm terminates once all shifts have been covered. By our previous inductive proof, we know that once the last shift in V' has been covered, all earlier shifts have been covered as well. We also know from our "greedy stays ahead" proof that an optimal solution could not have covered the last shift using fewer shifts than our greedy algorithm. Therefore, the greedy solution is also optimal (uses the smallest number of possible shifts).

4 The Economics of Heat Waves

You are an air conditioner mechanic, and your normally temperate coastal city is expecting record-breaking heat this summer. You have agreed to install or repair an air conditioning system for n clients, and each job will take one day to complete. You must decide the order in which to complete the jobs.

None of your clients want to be hot and uncomfortable in the coming heat wave, so they would all prefer to have their project completed earlier rather than later. In particular, after n days the uncomfortable heat is expected to be over, so nobody is going to be especially happy with being the last client on your list. More precisely, if you complete the job for client i at the end of day j, then client i's satisfaction is $s_i = n - j$.

Some clients are more important to you than others (perhaps they are paying you more, are more likely to recommend you to others, or are more likely to be repeat customers in the future). You assign each client i a weight w_i. You want to schedule your clients across the n days such that you maximize the weighed sum of satisfaction, that is, $\sum_{1 \le i \le n} w_i s_i$.

The following greedy strategy finds an optimal ordering of clients: Complete the projects in decreasing order of w_i.

1. **[5 marks]** Use an exchange argument to show that this greedy strategy yields an optimal schedule.

   Let $\mathcal{O}$ denote an optimal solution, with ordered weights given by $o_1, o_2, \ldots, o_n$. The weighted sum of satisfaction for $\mathcal{O}$ is $S(\mathcal{O}) = \sum_{i=1}^{n}(n-i)o_i$.

   Let p, q, where $p < q$, denote indices in $\mathcal{O}$ that define an inversion, relative to the order of the greedy ordering $\mathcal{G}$. That is, this is a pair of jobs that appear in a different order than they do in $\mathcal{G}$. Because the greedy strategy is to order jobs by decreasing weight, we must have $o_p \le o_q$. The weighted sum of satisfaction for $\mathcal{O}$ is

$$S(\mathcal{O}) = (n-1)o_1 + (n-2)o_2 + \ldots + (n-p)o_p + \ldots + (n-q)o_q + \ldots + (n-n)o_n.$$

   Let $\mathcal{O}'$ denote the solution obtained after we swap o_p with o_q in $\mathcal{O}$ (i.e., we swap the order so these two elements appear in the same order as they do in $\mathcal{G}$). Then

$$\begin{aligned} S(\mathcal{O}') &= (n-1)o_1 + (n-2)o_2 + \ldots + (n-p)o_q + \ldots + (n-q)o_p + \ldots + (n-n)o_n \\ &= S(\mathcal{O}) + (n-p)(o_q - o_p) + (n-q)(o_p - o_q) \\ &= S(\mathcal{O}) + (q-p)(o_q - o_p) \ge S(\mathcal{O}), \end{aligned}$$

   because $q > p$ and $o_q \ge o_p$. Therefore, the weighted satisfaction of $\mathcal{O}'$ is at least as high as the weighted satisfaction of $\mathcal{O}$.

   We can proceed in this way, repeatedly swapping inversions between the optimal schedule and the greedy schedule, until the optimal schedule is transformed into the greedy schedule. The weighted satisfaction will not decrease with each swap. Therefore, $S(\mathcal{G}) \ge S(\mathcal{O})$ and the greedy schedule is in fact optimal.

2. **[3 marks]** Instead of assuming that each job takes exactly 1 day, suppose that it takes t_i days to complete the job i (here t_i must be greater than 0, but is not necessarily an integer). Assume that $\sum_{1\le i\le n} t_i = n$. Again, you want to maximize the weighted sum of satisfaction $\sum_{1\le i\le n} w_i s_i$, where $s_i = n - c_i$. and c_i is the time to completion of the project. For example, if project i is completed first, then $c_i = t_i$. If a project i' is completed second (after project i), then $c_{i'} = t_i + t_{i'}$.

   Give and briefly explain a counterexample to show that the greedy strategy of completing projects in decreasing order of w_i is not optimal for this variant of the problem.

   Consider the example

i	w_i	t_i
1	5	2
2	3	1
3	2	0.5
4	1	0.5

   The greedy strategy would complete the projects in the order $[1, 2, 3, 4]$, with a total weighted satisfaction of

   $$5 * (4 - 2) + 3 * (4 - 3) + 2 * (4 - 3.5) + 1 * (4 - 4) = 13.$$

   A better strategy would be the ordering $[3, 2, 1, 4]$, which gives a total weighted satisfaction of

   $$2 * (4 - 0.5) + 3 * (4 - 1.5) + 5 * (4 - 3.5) + 1 * (4 - 4) = 17.$$

3. **[2 marks]** Describe a greedy strategy that is optimal on the problem variant described in part 2.

   An optimal greedy strategy is to perform the jobs in decreasing order of w_i/t_i.

4. **[5 marks]** Prove that your greedy strategy from part 3 is optimal.

   As in 4.1, let $\mathcal{O}$ denote an optimal schedule that completes jobs in the order $o_1, o_2, \ldots o_n$ (with respective weights $w_1, \ldots, w_n$ and completion times $t_1, \ldots, t_n$). And let p, q, where $p < q$ denote the indices of two jobs that appear in a different order in $\mathcal{O}$ than in the greedy ordering $\mathcal{G}$. For simplicity here in reasoning about the s_i values, we'll assume that p and q constitute a *neighbouring* inversion – that is, $q = p + 1$, and o_p and o_q appear in the opposite order in the greedy schedule. It's not hard to show that as long as a schedule has an inversion, it has a neighbouring inversion. We can easily show this by contradiction: if there are no neighbouring inversions in $\mathcal{O}$ – i.e., no pairs of consecutive elements that appear in a different order in $\mathcal{O}$ than in $\mathcal{G}$ – then it must mean there are no inversions.

   The weighted sum of satisfaction for $\mathcal{O}$ is

   $$S(\mathcal{O}) = w_1 s_1 + w_2 s_2 + \ldots + w_p s_p + w_q s_q + \ldots + w_n s_n,$$

   where $s_j = n - \sum_{i=1}^{j} t_i$. Let $\mathcal{O}'$ denote the schedule obtained by swapping jobs o_p and o_q in $\mathcal{O}$. The weighted satisfaction for $\mathcal{O}'$ is

   $$S(\mathcal{O}') = w_1 s_1 + w_2 s_2 + \ldots + w_q s'_q + w_p s'_p + \ldots + w_n s_n,$$

   where

   $$s'_q = n - (t_1 + t_2 + \ldots + t_{p-1} + t_q) = s_p + t_p - t_q$$

   and

   $$s'_p = n - (t_1 + t_2 + \ldots + t_{p-1} + t_q + t_p) = s_q.$$

Therefore,

$$\begin{aligned} S(\mathcal{O}') - S(\mathcal{O}) &= w_q s'_q + w_p s'_p - w_p s_p - w_q s_q \\ &= w_q s_p + w_q t_p - w_q t_q + w_p s_q - w_p s_p - w_q s_q \\ &= w_q (s_p - s_q + t_p - t_q) + w_p (s_q - s_p) \\ &= w_q t_p - w_p t_q, \end{aligned}$$

where we obtained the last equality by using $s_q = s_p - t_q$.

Because the greedy algorithm orders jobs in descending order by w_i/t_i, and jobs p and q appear in the opposite order in $\mathcal{G}$, we must have

$$\frac{w_q}{t_q} \geq \frac{w_p}{t_p},$$

which implies (by multiplying both sides by $t_p t_q$) that

$$w_q t_p \geq w_p t_q.$$

Thus, $S(\mathcal{O}') - S(\mathcal{O}) \geq 0$, and the weighted sum of satisfaction has not decreased by swapping elements p and q.

As mentioned earlier, as long as $\mathcal{O}$ has an inversion relative to the order of $\mathcal{G}$, it has a neighbouring inversion (i.e., a pair of two consecutive elements that appear in opposite order from $\mathcal{G}$). Thus, we can repeatedly swap neighbouring inversions until we have converted $\mathcal{O}$ to $\mathcal{G}$ without decreasing the weighted sum of satisfaction. From this we conclude that our greedy strategy yields an optimal schedule.