1 Unweighted Interval Scheduling

Suppose your friend has decided to change majors to one that grades based only on attendance. The only question is which classes to take next quarter? They all meet once a day at different times, *but are worth the same credits each*. Your friend's goal is to maximize the number of classes taken in the quarter without having to skip any lectures.

Problem Statement: we are given a set of n intervals, each of which has a start time s_i, a finish time f_i. Our goal is to select as large of a subset of the intervals such that no two selected intervals overlap.

We can solve this problem via dynamic programming; in fact, that technique is what we should use if the intervals have weights (classes worth differing numbers of credits). However, the solution is simpler if all intervals are worth the same value.

In fact, **one of the following algorithms will get the correct answer**. Decide which ones *don't work* by providing counter-examples. Don't worry (yet) about proving one that is correct.
Please note: for the homework and exam, you *do not* need to provide "not working" algorithms, and showing that other algorithms do not work *does not* demonstrate that yours does. The purpose of this part is to examine candidate algorithms and to think about the problem.

- Sign up for the class that begins earliest. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

- Sign up for the class that meets for the least amount of time. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

- Sign up for the class that conflicts with the fewest other classes. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

- Sign up for the class that ends earliest. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

Any of the above algorithms can be implemented in time $\mathcal{O}(n \log n)$ – a good exercise would be for you to write pseudo-code for it as part of your review. Unlike the dynamic programming algorithm, the correctness of this algorithm isn't as easy to see from the description. Right now, the "proof" that it is correct relies on that I told you one of the four was correct, and you've seen that the other three *aren't* correct. However, such a proof isn't valid.

Proving correctness: once we have an algorithm we believe is correct, we need to prove that it is. Each of the above algorithms can be described as "select some interval, remove conflicting intervals, and recursively solve the problem on the rest." We would like to prove that there is an optimal solution that includes the first interval selected.

Claim: There is an optimal solution that includes the first interval that we choose.
Note that I am not claiming that all optimal solutions do.

Would any optimal solution that includes our first interval also include any intervals that overlap with it?

What is left to do to prove that the rest of our algorithm is correct?

2 Interval Coloring

Suppose your friend is working at the library and is in charge of allowing groups into the various study rooms. For this problem, all study rooms are interchangable. A total of n groups have requested to use a study room tonight; group i would like to use it from s_i to f_i. If two groups overlap in their request times, they cannot be placed in the same study room; furthermore, we cannot reject a group. Fortunately, we have a very large library with an infinite number of study rooms, although we would prefer to not use all of them if possible.

Give an efficient algorithm that will assign each group to a room (the rooms are numbered $1, 2, 3, \ldots$) in such a way that the number of rooms you use is minimized, and no two groups that overlap are assigned to the same room. Explain as best you can why your algorithm achieves the optimal number of rooms (we'll talk about how to formally prove this).

Fractional Knapsack

Consider the general Knapsack problem: we have a set S of n items; each item has a positive benefit b_i and a positive weight w_i. We can carry at most some bound W, and we wish to take some (or all) of the items to gain the maximum possible benefit, subject to the constraint that the total weight we take is at most W.

During a discussion section during the Dynamic Programming portion of class, you saw an algorithm to solve a version of this problem called the **0-1 Knapsack Problem**. In this, each item had to either be taken or not taken; this represents that the items are things like your laptop, where having half a laptop is not worth half the value of a full laptop.

In today's problem, we can take a **fraction** of each item. That is, for each item, we decide some amount x_i to take, up to w_i. That is, we take up to the full amount of each item, which are presumed to be infinitely divisible – a not-perfect representation for things like food and beverage or the dust of valuable metals. This also assumes that the value is linear – having half of the available material is worth half the total value, one-quarter of the available material is worth one-quarter of its total value, and so on.

More formally, decide for each item a value x_i with $0 \leq x_i \leq w_i$ such that $\sum_i x_i \leq W$ with the goal of maximizing the sum $\sum_i b_i(x_i/w_i)$.

Example: Suppose we can carry $W = 10$ and have the following items available to us:

Item :	1	2	3	4	5
Weight:	4	8	2	6	1
Benefit:	12	32	40	30	50

Problem: find an efficient algorithm that will allow us to, given the general input, will find the optimal fractional knapsack solution. Prove that your algorithm finds the optimal amount.

3 Scheduling with Deadlines

Let's examine a different algorithm for scheduling intervals. In the last lecture, each interval had pre-designated start and end times. Instead, let's consider a problem where each interval i is a task that must be completed; each has a designated time t_i, but we can designate any start time for it. Each interval also has a deadline d_i, which can be different for each interval.

We must assign each interval a start time in such a way that no two intervals overlap. Ideally, we would like to schedule everything to be finished before its deadline, but this is not always possible. We say the lateness l_i of a job is how late it is finished compared to its deadline, $s_i + t_i - d_i$, or 0 if it has been completed by the deadline. Our goal is to minimize the *maximum* lateness: the amount by which the most late job exceeds its deadline.

Examples: What is the optimal schedule for each of the following?

Example 1:

Time	1	2	3
Deadline	2	4	6

Example 2:

Time	1	2	3	4
Deadline	2	4	6	6

Some Possible Algorithms One of the following algorithms will correctly schedule the tasks. Decide which one you think it is, and show that the other two *do not* correctly schedule these.

- Sort the jobs by increasing time t_i; schedule them in that order.
- Sort the jobs by $d_i - t_i$; schedule them in that order.
- Sort the jobs by deadline d_i; schedule them in that order.

Proof of Correctness

Since every schedule (optimal or otherwise) includes every task, we cannot follow the same model proof as the covering/packing problems from last lecture. Instead, I will first cover two lemmas[1], and then I will use those to prove the overall theorem.

Lemma 1. When deciding start times, don't leave any gaps; $s_{i+1} = s_i + t_i$.

Lemma 2. Any schedule that doesn't agree with our algorithm has at least one pair of *consecutive* intervals $i, i+1$ that are *inverted* relative to our order.

We can now proceed to the full proof; we claim our output is a global optimal with this claim:

Claim: Any schedule with an inversion (relative to our output) can be modified to be more like our algorithm's output without making it worse.

[1] You can think of a lemma as a "helper proof" – a statement that requires a proof for itself, but the overall statement is then used in your proof.