CS331: Algorithms and Complexity
Part IV: Greedy Algorithms

Kevin Tian

1 Introduction

In Part III of the notes, we saw how DP could be a powerful tool in algorithms requiring repeated decisions. For example, in the *unbounded knapsack* problem (Section 3.3, Part III), we were given the weights W and values V of n items, as well as a budget B. The problem required us to find the count vector $\mathbf{c} \in \mathbb{Z}^n_{\geq 0}$ maximizing the total item value, i.e., $\sum_{i\in[n]} \mathbf{c}_i V[i]$, while staying within the budget, i.e., $\sum_{i\in[n]} \mathbf{c}_i W[i] \leq B$. Our strategy was to use DP to decide which items to take, by considering $O(n)$ candidate subproblems at each step and using the best option recursively, taking care to memoize solutions. This gave an $O(nB)$-time algorithm overall.

This seems like a lot of work for a problem which admits an intuitive heuristic: choose the most valuable item every time. However, this heuristic can get us into trouble, since earlier decisions can later limit our options. For example, if $W = \{1, 3\}$, $V = \{2, 3\}$, and $B = 4$, choosing items by value fails to solve the problem: we can achieve a value of 8 by taking the first item four times (as $W[1] = 1$), whereas taking the second, more valuable, item limits our achievable value at 5.

You might complain, the first item is "obviously" better than the second: while less valuable, it is more valuable per unit of weight. This suggests choosing items ordered by their *value density* $\frac{V[i]}{W[i]}$, taking the most value-dense item first. This is also not optimal: if $W = \{2, 3\}$, $V = \{4, 7\}$, and $B = 4$, the second item is more value-dense ($\frac{7}{3} > \frac{4}{2}$), but taking it limits our maximum achievable value to 7, whereas obtaining value 8 (by taking the first item twice) is optimal.

Fortunately, under a slight modification to unbounded knapsack, our heuristic succeeds: allowing for fractional counts. Here, we imagine each item is a divisible liquid, so we can take e.g., 1.5 units of an item. Formally, in the *fractional unbounded knapsack* problem, the setting is the same as unbounded knapsack, except we allow for fractional $\mathbf{c} \in \mathbb{R}^n_{\geq 0}$, rather than forcing $\mathbf{c} \in \mathbb{Z}^n_{\geq 0}$. Our goal is to compute the maximum possible value $\sum_{i\in[n]} \mathbf{c}_i V[i]$ subject to $\sum_{i\in[n]} \mathbf{c}_i W[i] \leq B$. We claim that the following simple $O(n)$-time algorithm solves this problem optimally.

Algorithm 1: FracUnboundedKnapsack(W, V, B)

Input: W, V, two Array instances containing n numbers in $\mathbb{R}_{>0}$, $B > 0$
$i^\star \leftarrow \text{argmax}_{i\in[n]} \frac{V[i]}{W[i]}$
return $V[i^\star] \cdot \frac{B}{W[i^\star]}$

Algorithm 1 says to take only the most value-dense item. It is a *greedy algorithm*: an algorithm which makes its decisions according to a pre-specified rule (e.g., maximum value density), rather than via recursion and memoization, as in DP. There are many appealing properties of greedy algorithms: they are straightforward to state, easy to implement efficiently, and intuitive.

On the other hand, one must take care when using greedy algorithms: they can often be incorrect if the wrong selection rule is chosen. In fact, oftentimes there is no natural correct greedy rule at all. For example, how do we know that, despite the failure of greedy algorithms for unbounded problem, they actually succeed in the fractional variant? We gave a counterexample showing that greedy is not optimal with integer counts, but how would one prove that it actually is always optimal when fractional counts are allowed? These notes develop general guidelines and techniques for designing greedy algorithms, and proving their correctness.

2 Rearrangement

One of the most powerful tools for arguing about the optimality of greedy algorithms is the *rearrangement lemma* (Lemma 1). This lemma is quite intuitive: it says that if you have two "amounts of items to take" a_1 and a_2, and two "item densities" b_1 and b_2, if your goal is to take the most total weight, you should take more of the larger-density item. It turns out this lemma is true even when all of these values are fractional or even negative, making it very simple to remember.

Lemma 1 (Rearrangement lemma)**.** *Suppose $a_1 \geq a_2$ and $b_1 \geq b_2$. Then $a_1b_1 + a_2b_2 \geq a_1b_2 + a_2b_1$.*

Proof. It suffices to expand: $(a_1 - a_2)(b_1 - b_2) \geq 0 \implies (a_1b_1 + a_2b_2) - (a_1b_2 + a_2b_1) \geq 0$. □

The main message of Lemma 1 is straightforward, yet it and its variations are surprisingly powerful in proving optimality of greedy algorithms. We now give several examples of its use.

2.1 Fractional unbounded knapsack

Let us revisit the fractional unbounded knapsack example. We first simplify slightly: nothing changes if the i^{th} item instead has weight 1 and value $D[i] = \frac{V[i]}{W[i]}$, for all $i \in [n]$ (the value density). This is because we can take arbitrary item amounts, and $\mathbf{c}_i$ units of the original item is equivalent to $\mathbf{c}_i W[i]$ units of the new item, as both yield $\mathbf{c}_i V[i] = (\mathbf{c}_i W[i]) \cdot D[i]$ value and $\mathbf{c}_i W[i] = (\mathbf{c}_i W[i]) \cdot 1$ weight. Our new problem is

$$\max_{\mathbf{c} \in \mathbb{R}^n_{\geq 0}} \sum_{i \in [n]} \mathbf{c}_i \cdot D[i] \text{ subject to } \sum_{i \in [n]} \mathbf{c}_i \leq B. \tag{1}$$

Let $i^\star \in [n]$ be the index of the item with the largest value density, computed in Line 2 of Algorithm 1. Thus, Algorithm 1 returns the solution to (1) with $\mathbf{c}_{i^\star} = B$ and $\mathbf{c}_i = 0$ for all $i \neq i^\star$. Denote this solution, returned by Algorithm 1, by $\mathbf{c}^{\text{alg}}$. We claim it is optimal for (1).

To see this, let $\mathbf{c}^{\text{opt}} \in \mathbb{R}^n_{\geq 0}$ optimally solve (1). We transform it into $\mathbf{c}^{\text{alg}}$ and show that our transformation cannot decrease the value of (1), which implies $\mathbf{c}^{\text{alg}}$ is also optimal for (1).

We now give this transformation. Take any $i \in [n]$ where $i \neq i^\star$. Moving all $\mathbf{c}^{\text{opt}}_i$ units of weight from i to $i^\star$ cannot decrease (1), by Lemma 1 with $a_1 = \mathbf{c}^{\text{opt}}_i$, $a_2 = 0$, $b_1 = D[i^\star]$, and $b_2 = D[i]$. Repeating for each $i \in [n]$ transforms $\mathbf{c}^{\text{opt}}$ into a vector that puts all $\sum_{i \in [n]} \mathbf{c}^{\text{opt}}_i \leq B$ units of weight on item $i^\star$. Thus, it achieves less value in (1) than $\mathbf{c}^{\text{alg}}$, which puts all B units of weight allowed on item $i^\star$, since $D[i^\star] > 0$ by assumption. This proves $\mathbf{c}^{\text{alg}}$ is optimal, as claimed.

This example illustrates a general strategy in greedy algorithms. While we do not know what $\mathbf{c}^{\text{opt}}$ is, an optimal solution to (1) certainly *exists*, so it makes sense as a benchmark. On the other hand, we know exactly what the solution $\mathbf{c}^{\text{alg}}$ is, because we defined an algorithm to produce it. The rest of the argument gradually shows via a transformation that whatever the vector $\mathbf{c}^{\text{opt}}$ is, our algorithm's $\mathbf{c}^{\text{alg}}$ attains at least as good of an objective value in (1), so it is also optimal.

2.2 Total completion time

Lemma 1 generalizes to n variables. Consider the following situation: you now have n "amounts of items to take" in sorted order $\mathbf{a}_1 \geq \mathbf{a}_2 \geq \ldots \geq \mathbf{a}_n$, as well as n "item densities" in sorted order $\mathbf{b}_1 \geq \mathbf{b}_2 \geq \ldots \geq \mathbf{b}_n$. You are required to take $\mathbf{a}_1$ copies of some item (not necessarily the item with density $\mathbf{b}_1$), $\mathbf{a}_2$ copies of another item (not necessarily the item with density $\mathbf{b}_2$), and so on, with the goal of maximizing the total weight. Corollary 1 claims that the optimal way to do this is to take $\mathbf{a}_1$ copies of the densest item, $\mathbf{a}_2$ copies of the second most dense, and so forth. It also states that the worst way to do this is by reversing the order of items taken.

Corollary 1 (Rearrangement inequality)**.** *Let $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ have nondecreasing coordinates, i.e., $\mathbf{a}_1 \geq \mathbf{a}_2 \geq \ldots \geq \mathbf{a}_n$ and $\mathbf{b}_1 \geq \mathbf{b}_2 \geq \ldots \geq \mathbf{b}_n$. Let $\pi : [n] \to [n]$ be a permutation.*[1] *Then,*

$$\sum_{i \in [n]} \mathbf{a}_i \mathbf{b}_i \geq \sum_{i \in [n]} \mathbf{a}_i \mathbf{b}_{\pi(i)} \geq \sum_{i \in [n]} \mathbf{a}_i \mathbf{b}_{n+1-i}.$$

[1] A permutation is a one-to-one function from $[n] \to [n]$: if $n = 3$, $\pi(1) = 2$, $\pi(2) = 1$, $\pi(3) = 3$ is a permutation.

Proof. Suppose for the sake of contradiction that the first inequality above is false. Then π cannot be the identity permutation (which sets $\pi(i) = i$ for all $i \in [n]$), so there is at least one *inversion* $(i, j) \in [n] \times [n]$, with $\pi(i) > \pi(j)$ but $i < j$. Swapping $\pi(i)$ and $\pi(j)$ can never decrease our value $\sum_{i \in [n]} \mathbf{a}_i \mathbf{b}_{\pi(i)}$, by Lemma 1. Repeating until π agrees with the identity gives a contradiction.

To see the second inequality above, we repeat the same argument but now our goal is to turn any non-inversion into an inversion. Lemma 1 shows this decreases objective value, and the only permutation such that every pair of indices is inverted is the reverse of the identity. □

We now give an application of Corollary 1 to the *total completion time* problem. In this problem, we are given as input T, an Array instance containing n positive numbers representing the durations of n jobs we wish to complete, so the i^{th} job has a duration of $T[i]$. We are tasked with assigning intervals $[s_i, e_i] \subset \mathbb{R}_{\geq 0}$ such that $e_i - s_i = T[i]$ (i.e., enough time has passed that we can service the i^{th} job), and no two intervals overlap. Our goal is to assign intervals such that

$$\sum_{i \in [n]} e_i, \tag{2}$$

i.e., the sum of all completion times, is minimized. One motivation for this problem is to maximize the total "utility" of all jobs. Another perspective is provided by Chapter 4.1, [Eri24]: if we treat the durations $T[i]$ as the lengths of files to be stored on a tape, and we must scan through the first e_i addresses to finish accessing the i^{th} file, then minimizing $\frac{1}{n} \sum_{i \in [n]} e_i$ (which is the same as minimizing (2)) optimizes the average access time of the n files.

We first observe that the optimal choice of intervals spends no *idle time*: it begins the $(i+1)^{\text{th}}$ job immediately after finishing the i^{th} job. Otherwise, removing idle intervals can only decrease the completion objective (4), since all e_i get smaller. Therefore, the optimal solution has the structure: for some permutation $\pi : [n] \to [n]$,

$$e_{\pi(i)} = \sum_{j \in [i]} T[\pi(j)]. \tag{3}$$

In other words, we schedule jobs back-to-back, one at a time according to the permutation π (so $\pi(1)$ is the index of the first job serviced, $\pi(2)$ is the second, and so on), so that the $\pi(i)^{\text{th}}$ job is completed at a time $e_{\pi(i)}$ when all earlier jobs (according to π) have finished.

The only degree of freedom in our solution is now the choice of permutation π. We hence rewrite the problem as

$$\min_{\pi:[n]\to[n] \text{ is a permutation}} f(\pi), \text{ where } f(\pi) := \sum_{i \in [n]} \sum_{j \in [i]} T[\pi(j)]. \tag{4}$$

We claim that the optimal π is the one that sorts T in nondecreasing order, i.e., we service the shortest job first. Assume for simplicity that T has already been sorted in this way, so our goal is to prove that the identity π is optimal for (4). We prove this by rewriting (4) in a simpler way:

$$\begin{aligned} f(\pi) = \sum_{j \in [i]} T[\pi(j)] &= T[\pi(1)] + (T[\pi(1)] + T[\pi(2)]) + (T[\pi(1)] + T[\pi(2)] + \ldots + T[\pi(n)]) \\ &= n \cdot T[\pi(1)] + (n-1) \cdot T[\pi(2)] + \ldots + T[\pi(n)] \\ &= \sum_{i \in [n]} (n + 1 - i) \cdot T[\pi(i)]. \end{aligned} \tag{5}$$

In other words, we need to pay some duration n times, another duration $n - 1$ times, and so on. Now the claim that the identity permutation is optimal follows from Corollary 1, applied with $\mathbf{a}_i = T[i]$ and $\mathbf{b}_i = i$ for all $i \in [n]$. In particular, $\mathbf{a}$ and $\mathbf{b}$ are sorted in reverse order to each other, so Corollary 1 shows that the minimal way to pair up their coordinates is by reversing the order of $\mathbf{b}$, which is exactly achieved by (5) with the identity permutation $\pi(i) = i$.

2.3 Fractional 0-1 knapsack

We finally consider a variant of the fractional unbounded knapsack problem: the *fractional* 0-1 *knapsack problem*, where the goal is to solve

$$\max_{\mathbf{c}\in[0,1]^n} f(\mathbf{c}) \text{ subject to } \sum_{i\in[n]} \mathbf{c}_i W[i] \le B, \text{ where } f(\mathbf{c}) := \sum_{i\in[n]} \mathbf{c}_i V[i]. \tag{6}$$

Note that in (6), the count vector $\mathbf{c}$ has $\mathbf{c} \in [0,1]^n$, i.e., we are still allowed to take fractional amounts, but we cannot take more than one unit of any item. We make the simplifying assumption that all value densities $\frac{V[i]}{W[i]}$ are distinct, since otherwise we can lump together all of the items with the same density into a single item, which does not change the problem.

This problem appears more challenging than either of those we previously handled. Unlike the unbounded variant in Section 2.1, we cannot simplify so that all weights are identical, because the amount of each item that is available to us matters. Moreover, the amounts we can take change depending on which items are taken, so Corollary 1 does not quite apply either. Nonetheless, you may guess that we should prioritize taking value-dense items first, just like in the unbounded case. Indeed, we claim that the following greedy Algorithm 2 solves this problem optimally.

Algorithm 2: FracZOKnapsack(W, V, B)

```
Input: W, V, two Array instances containing n numbers in ℝ_{>0}, B > 0
Sort W, V similarly, so that items i ∈ [n] are in decreasing order by V[i]/W[i]
(b, v, i) ← (B, 0, 1)
while b − W[i] ≥ 0 and i ∈ [n] do
    (b, v, i) ← (b − W[i], v + V[i], i + 1)
end
if i == n + 1 then
    return v
end
else
    return v + (b / W[i]) · V[i]
end
```

Algorithm 2 first sorts the items by value density, so that the most value-dense item comes first. It then repeatedly takes entire items until either taking the next item would go over the weight budget, or all items are taken. In the former case, it takes as much of the last item as possible.

Assume that $\sum_{i\in[n]} W[i] > B$, as otherwise Algorithm 2, which produces the all-ones vector in this case, is clearly optimal (no feasible $\mathbf{c}$ can be larger than 1 entrywise). Let $i^\star \in [n]$ be the value of i when Algorithm 2 terminates. Algorithm 2 then attains value $f(\mathbf{c}^{\text{alg}})$ for (6), where

$$\mathbf{c}_i^{\text{alg}} = \begin{cases} 1 & i \in [i^\star - 1] \\ \frac{B - \sum_{i\in[i^\star-1]} W[i]}{W[i^\star]} & i = i^\star \\ 0 & i \in [n] \setminus [i^\star] \end{cases}. \tag{7}$$

Suppose for the sake of contradiction that $\mathbf{c}^{\text{alg}}$ is not optimal for (6). Let $\mathbf{c}^{\text{opt}} \in [0,1]^n$ instead be optimal for (6). We obtain a contradiction by transforming $\mathbf{c}^{\text{opt}}$ into a different vector $\mathbf{c}'$ achieving a strictly greater total value, i.e., $f(\mathbf{c}') \ge f(\mathbf{c}^{\text{opt}})$, without affecting the total weight taken.

To do so, we claim that there are indices j, k with $1 \le j < k \le n$ such that $\mathbf{c}_j^{\text{opt}} < \mathbf{c}_j^{\text{alg}}$ and $\mathbf{c}_k^{\text{opt}} > \mathbf{c}_k^{\text{alg}}$. Assuming this is true, the idea is to slightly shift some weight from item k to item j (which has higher value density), so that the budget is not violated but we have strictly greater value. This contradicts the assertion that $\mathbf{c}^{\text{opt}} \ne \mathbf{c}^{\text{alg}}$ is optimal.

More formally, let $\Delta := \min(\mathbf{c}_k^{\text{opt}} - \mathbf{c}_k^{\text{alg}}, \frac{W[j]}{W[k]} \cdot (\mathbf{c}_j^{\text{alg}} - \mathbf{c}_j^{\text{opt}})) > 0$, and define

$$\mathbf{c}'_i = \begin{cases} \mathbf{c}_j^{\text{opt}} + \Delta \cdot \frac{W[k]}{W[j]} & i = j \\ \mathbf{c}_k^{\text{opt}} - \Delta & i = k \\ \mathbf{c}_i^{\text{opt}} & i \notin \{j, k\} \end{cases}.$$

That is, the only coordinates of $\mathbf{c}^{\text{opt}}$ that changed were the j^{th} and k^{th} coordinates, and by our choice of Δ, we still have $\mathbf{c}' \in [0,1]$ since $\mathbf{c}'_k \geq \mathbf{c}^{\text{alg}}_k$ and $\mathbf{c}'_j \leq \mathbf{c}^{\text{alg}}_j$. The total weight change is

$$\left(\Delta \cdot \frac{W[k]}{W[j]}\right) \cdot W[j] - \Delta \cdot W[k] = 0,$$

so $\mathbf{c}'$ stays within the budget as it achieves the same total weight as $\mathbf{c}^{\text{opt}}$. However, $\mathbf{c}'$ yields a higher objective value than $\mathbf{c}^{\text{opt}}$ in (6), since the j^{th} item has a higher value density than the k^{th}:

$$f(\mathbf{c}') - f(\mathbf{c}^{\text{opt}}) = \left(\Delta \cdot \frac{W[k]}{W[j]}\right) \cdot V[j] - \Delta \cdot V[k] = \left(\frac{V[j]}{W[j]} - \frac{V[k]}{W[k]}\right) \cdot \Delta \cdot W[k] > 0.$$

Therefore, as long as such indices j, k exist, no other count vector $\mathbf{c}^{\text{opt}}$ can be optimal.

We conclude by proving our earlier claim about the existence of j, k. Let j be the first index where $\mathbf{c}^{\text{alg}}_j \neq \mathbf{c}^{\text{opt}}_j$. By construction, $\mathbf{c}^{\text{opt}}_j < \mathbf{c}^{\text{alg}}_j$, since $\mathbf{c}^{\text{alg}}$ puts the maximum possible count on each index sequentially. So, there must be $k > j$ where $\mathbf{c}^{\text{opt}}_k > \mathbf{c}^{\text{alg}}_k$. Otherwise, $\mathbf{c}^{\text{opt}}_k \leq \mathbf{c}^{\text{alg}}_k$ for all $k \in [n]$, and hence $\mathbf{c}^{\text{opt}}$ cannot attain greater value in (6), contradicting its definition.

3 Exchange arguments

The examples developed in Section 2 follow a similar structure. In each case, our goal was to argue that a solution x^{alg} produced by a greedy algorithm was optimal for some objective function f. For example, in Section 2.2 we reduced the problem to the form (4), an optimization problem in an objective f over permutations $\pi : [n] \to [n]$, and claimed that the permutation π^{alg} sorting jobs in nondecreasing duration order was optimal. Similarly, in Section 2.3, we claimed that $\mathbf{c}^{\text{alg}}$ in (7) was optimal among $\mathbf{c} \in [0,1]^n$ meeting a weight constraint, for the objective f in (6).

We proved optimality of the solutions x^{alg} in each case by first assuming the existence of an optimal solution, x^{opt}, and comparing $f(x^{\text{opt}})$ to $f(x^{\text{alg}})$. Of course, our goal is to establish that $f(x^{\text{alg}}) \geq f(x^{\text{opt}})$ (or vice versa, if our objective is minimization of f rather than maximization). We saw a few different ways to prove this type of bound.

In Section 2.1, we gradually transformed x^{opt} into x^{alg}, and showed f only improved after each step of the transformation. In Section 2.2, we applied Corollary 1, which is really just an iterative application of Lemma 1 that transforms a permutation into the optimal permutation. Finally, in Section 2.3, we showed how any solution unequal to our greedy algorithm's choice could be transformed into a better solution, preventing any other solution from being optimal.

These are examples of *exchange arguments* that transform a purported optimal solution x^{opt} into our solution x^{alg}. These exchange-based transformations can be partial or full, as well as piece-by-piece or all-at-once, but in each case the goal is to argue that the greedy solution x^{alg} does better than any other candidate x^{opt}. One can either achieve this by proving that f continually does not get worse until our transformation is complete (as done in Sections 2.1 and 2.2), in which case we have shown that x^{alg} indeed better candidate than x^{opt}, or by directly contradicting the definition of x^{opt} by demonstrating a strict objective value improvement (as done in Section 2.3). We give several additional examples of exchange arguments in this section.

3.1 Weighted total completion time

We first consider a weighted variant of the total completion time problem from Section 2.2. Here, in addition to the input Array T containing the duration times of n jobs, we are also given W, an Array instance containing n positive numbers representing the weights of our jobs. We are tasked with assigning non-overlapping intervals $[s_i, e_i = s_i + T[i]] \subset \mathbb{R}_{\geq 0}$, with the goal of minimizing

$$\sum_{i \in [n]} e_i W[i].$$

That is, we care about minimizing the weighted sum of completion times.

As before, clearly there is no advantage to idle time, so the optimal solution is to choose a permutation $\pi : [n] \to [n]$ such that the completion time relationship (3) holds. Following (4), our new optimization problem is:

$$\min_{\pi:[n]\to[n] \text{ is a permutation}} f(\pi), \text{ where } f(\pi) := \sum_{i\in[n]} W[\pi(i)] \left(\sum_{j\in[i]} T[\pi(j)] \right). \tag{8}$$

We claim that sorting W, T similarly such that $\frac{T[i]}{W[i]}$ is in nondecreasing order is optimal. Assume that the lists have already been sorted in this way, in which case our claim is that the identity permutation π is optimal for (8). Our proof will not be quite as straightforward as in Section 2.2, since Corollary 1 does not apply directly. This is because the choice of π dictates the weight of each job; unlike in (5), we are not just taking n copies of some duration $T[\pi(1)]$, $n-1$ copies of some other duration, and so on, as the number of copies changes depending on π.

Nonetheless, we can use a similar strategy as in the proof of Corollary 1. Recall that the identity permutation is the unique permutation without inversions. We would like to show that undoing inversions in π decreases the value of $f(\pi)$, the same idea that was used to prove Corollary 1. However, this strategy seems more complicated here, due to the fact that swapping two intervals affects the completion times of all intervals in between, significantly altering the formula (8).

We fix this issue with a stronger observation about non-identity permutations.

Lemma 2. *Let $\pi : [n] \to [n]$ be a non-identity permutation. Then π has an* adjacent inversion, *i.e., a pair of indices $(i, i+1)$ such that $\pi(i) > \pi(i+1)$.*

Proof. Suppose π has no adjacent inversion. Then $\pi(1) \le \pi(2) \le \ldots \le \pi(n)$. Because π is a permutation, the only possible π satisfying these inequalities is the identity, a contradiction. □

Lemma 2 dramatically simplifies our argument: by undoing only adjacent inversions, we localize the change in the objective. Namely, swapping two adjacent intervals $\pi(i)$ and $\pi(i+1)$ does not affect the completion time of any other interval, so only two terms in the formula (8) change. We are now ready to prove that the identity permutation is optimal for (8) via this approach.

Suppose that π is optimal for (8), and suppose it is not the identity permutation. By Lemma 2, there is an adjacent inversion $(i, i+1)$, i.e., $\pi(i) > \pi(i+1)$ for some $i \in [n-1]$. Because we assumed indices were sorted by $\frac{T[i]}{W[i]}$, $\pi(i) > \pi(i+1)$ means that

$$\frac{T[\pi(i)]}{W[\pi(i)]} \ge \frac{T[\pi(i+1)]}{W[\pi(i+1)]}. \tag{9}$$

Let π' be identical to π, except $\pi'(i+1) = \pi(i)$ and $\pi'(i) = \pi(i+1)$, i.e., it swaps the i^{th} and $(i+1)^{\text{th}}$ intervals and leaves the order unchanged otherwise. We compute

$$\begin{aligned}
f(\pi) - f(\pi') &= \sum_{i\in[n]} W[\pi(i)] \left(\sum_{j\in[i]} T[\pi(j)] \right) - \sum_{i\in[n]} W[\pi'(i)] \left(\sum_{j\in[i]} T[\pi'(j)] \right) \\
&= W[\pi(i+1)] \left(\sum_{j\in[i+1]} T[\pi(j)] \right) + W[\pi(i)] \left(\sum_{j\in[i]} T[\pi(j)] \right) \\
&\quad - W[\pi(i+1)] \left(\sum_{j\in[i-1]} T[\pi(j)] + T[\pi(i+1)] \right) - W[\pi(i)] \left(\sum_{j\in[i+1]} T[\pi(j)] \right) \\
&= W[\pi(i+1)]T[\pi(i)] - W[\pi(i)]T[\pi(i+1)] \ge 0,
\end{aligned}$$

where in the last line we used (9). Thus, undoing an adjacent inversion cannot increase the objective value. After undoing all adjacent inversions, the contrapositive of Lemma 2 shows that we end up at the identity permutation, which shows the identity is optimal for (8) as claimed.

3.2 Minimizing lateness

We next consider a related problem where a similar strategy applies. In the *minimizing lateness* problem, our task is again to schedule n jobs, with positive durations given by an input Array T, into non-overlapping intervals $[s_i, e_i = s_i + T[i]] \subset \mathbb{R}_{\geq 0}$. We are given an additional input another Array instance D containing n positive numbers, representing job deadlines. Our goal is to assign intervals that minimize the maximum lateness of any of our jobs: $\max_{i \in [n]} e_i - D[i]$.

It is clear that idle time can only increase the maximum lateness, so we again rephrase the problem in terms of permutations. As before, if we schedule the jobs back-to-back according to a permutation π, the completion times are given by (3). Thus, our goal is to compute

$$\min_{\pi:[n]\to[n] \text{ is a permutation}} f(\pi), \text{ where } f(\pi) := \max_{i \in [n]} e_{\pi(i)} - D[\pi(i)] = \max_{i \in [n]} \sum_{j \in [i]} T[\pi(j)] - D[\pi(i)]. \quad (10)$$

We claim that if T, D are sorted similarly such that D is in nondecreasing order, the identity permutation π is optimal for (10). We prove this using a similar argument as in Section 3.1.

Again suppose π is not the identity, so Lemma 2 shows π has an adjacent inversion $(i, i+1)$, i.e., $\pi(i) > \pi(i+1)$. By assumption, this means $D[\pi(i)] \geq D[\pi(i+1)]$. Let π' be the permutation that swaps $\pi(i)$ and $\pi(i+1)$, leaving all other entries unchanged. We claim $f(\pi') \leq f(\pi)$.

Let j be the maximizing argument in (10) for $f(\pi')$, i.e.,

$$f(\pi') = e_{\pi'(j)} - D[\pi'(j)]$$

is the lateness of the j^{th} interval. If $j \notin \{i, i+1\}$, then $\pi'(j) = \pi(j)$ and $e_{\pi'(j)} = e_{\pi(j)}$, so that

$$f(\pi') = e_{\pi'(j)} - D[\pi'(j)] = e_{\pi(j)} - D[\pi(j)] \leq \max_{k \in [n]} e_{\pi(k)} - D[\pi(k)] = f(\pi).$$

In other words, swapping i, i' only creates more opportunities to be later than the j^{th} interval.

This leaves the case where $j \in \{i, i+1\}$. To handle this case it suffices to show

$$\begin{aligned} &\max\left(A + T[\pi(i+1)] - D[\pi(i+1)],\ A + T[\pi(i)] + T[\pi(i+1)] - D[\pi(i)]\right) \\ &\leq \max\left(A + T[\pi(i)] - D[\pi(i)],\ A + T[\pi(i)] + T[\pi(i+1)] - D[\pi(i+1)]\right), \end{aligned} \quad (11)$$

where we denote the sum of durations of the jobs completed before $\pi(i)$ or $\pi(i+1)$ by

$$A := \sum_{j \in [i-1]} T[\pi(j)].$$

In other words, (11) asks us to show the new maximum lateness between the intervals $i, i+1$ is at most the previous maximum lateness between these intervals.

Finally, (11) is true since its right-hand side's second term dominates both terms on the left-hand side, using our assumption that $D[\pi(i)] \geq D[\pi(i+1)]$, because i was an adjacent inversion:

$$\begin{aligned} A + T[\pi(i+1)] - D[\pi(i+1)] &\leq A + T[\pi(i)] + T[\pi(i+1)] - D[\pi(i+1)], \\ A + T[\pi(i)] + T[\pi(i+1)] - D[\pi(i)] &\leq A + T[\pi(i)] + T[\pi(i+1)] - D[\pi(i+1)]. \end{aligned}$$

We have thus shown using (11) that undoing adjacent inversions improves our objective (10). Repeatedly undoing such inversions results in the identity permutation, which is hence optimal.

4 Greedy stays ahead

Thus far, our proofs of optimality have relied on exchange arguments: repeatedly finding pairs of choices to exchange that result in any purportedly optimal solution eventually becoming our greedy solution, while improving objectives at each step. While this idea is straightforward, applying it in different scenarios can take some ingenuity (e.g., should we apply Corollary 1 "all at once," or should we instead try to make local improvements based on adjacent inversions)?

In this section, we develop an alternative proof strategy introduced in [KT05], Chapter 4 as the "greedy stays ahead" argument. This strategy is much more formulaic to apply, yet captures the optimality of a wide range of greedy algorithms. The idea is to inductively show that if an algorithm must make n choices sequentially, then after any number of $i \in [n]$ choices, the greedy algorithm has done the best it could up to that point (compared to any other algorithm that has also made i choices). This "staying ahead" property will then let us continue our induction, and once all n choices have been made, we can conclude the greedy algorithm is optimal.

We give several examples that formalize this idea in this section.

4.1 Scheduling revisited

To begin, we revisit the scheduling problem from Section 3.1, Part III. The setup is exactly the same as before: we wish to schedule a maximum-size set of non-overlapping intervals, given as L, an Array of n tuples (ℓ_i, r_i). In other words, we wish to find

$$\max_{S \subseteq [n]} f(S) \text{ subject to } S \text{ is non-overlapping, where } f(S) := |S|.$$

We claim that the following greedy Algorithm 3 succeeds in solving this problem.

Algorithm 3: GreedyScheduling(L)

```
1 Input: L, an Array instance containing n tuples {(ℓ_i, r_i)}_{i∈[n]} in ℝ^2 with ℓ_i < r_i for all i ∈ [n]
2 Sort L in non-decreasing order by r_i, i.e., r_1 ≤ r_2 ≤ ... ≤ r_n
3 (count, i) ← (1, 1)
4 for 2 ≤ j ≤ n do
5 |   if ℓ_j > r_i then
6 |   |   (count, i) ← (count + 1, j)                    // Include interval j.
7 |   end
8 end
9 return count
```

Algorithm 3 maintains a current index i, pointing to the last interval included in the set S. It scans through the intervals in L, trying to find the first index j where $\ell_j > r_i$, so that the j^{th} interval does not overlap with the i^{th}. It includes the first such interval it finds, updating i and the count appropriately. The algorithm terminates after we have looped through all of L.

We prove optimality of Algorithm 3 using a "greedy stays ahead" argument. To motivate it, we think of any scheduling algorithm as making a sequence of choices: which indices to include, one by one. Without loss of generality, we choose these indices in increasing order according to their right endpoint. The idea is to show that after any i choices, the last interval chosen by Algorithm 3 stays ahead of the last interval chosen by any other algorithm.

We now formalize these ideas. Let $S^{\text{alg}} \subseteq [n]$ be the indices included by Line 6 of Algorithm 3. We denote these indices $S^{\text{alg}} = \{a_1, a_2, \ldots, a_k\}$, sorted so that $a_1 < a_2 < \ldots < a_k$.

To implement the argument, let S^{opt} be an optimal set of non-overlapping interval indices, denoted $S^{\text{opt}} := \{b_1, b_2, \ldots, b_m\}$ sorted so that $b_1 < b_2 < \ldots < b_m$. By optimality of S^{opt}, and because S^{alg} is a non-overlapping set by construction, we must have $m = |S^{\text{opt}}| \geq |S^{\text{alg}}| = k$. Our main claim is that for all $i \in [k]$, $a_i \leq b_i$, or equivalently the i^{th} interval in S^{alg} ends before the i^{th} interval in S^{opt} ends, because we sorted L in non-decreasing order by right endpoint. This is the sense in which the greedy solution from Algorithm 3 stays ahead of any other solution.

We prove $a_i \leq b_i$ via induction. Because we sorted by right endpoint, an equivalent claim is $r_{a_i} \leq r_{b_i}$. For the base case, Algorithm 3 always sets $a_1 = 1$ (taking the first interval), so $a_1 \leq b_1$.

Next, suppose that $a_i \leq b_i$ for some $i \in [n]$. Since the b_{i+1}^{th} interval does not overlap with the b_i^{th} interval, we have $\ell_{b_{i+1}} > r_{b_i}$. However, by induction this implies $\ell_{b_{i+1}} > r_{a_i}$. This means that b_{i+1} was a valid interval to include in S^{alg} after taking our first i choices, as it begins after the interval $[\ell_{a_i}, r_{a_i}]$ ends. Because Algorithm 3 includes the index of the first available interval after $[\ell_{a_i}, r_{a_i}]$ ends as a_{i+1}, we can conclude the desired $a_{i+1} \leq b_{i+1}$.

Let us see why this claim implies $k = m$. Suppose for contradiction that $m > k$, and consider the subset $S_{\text{alg}} \cup \{b_{k+1}\}$. We claim that this is also a non-overlapping subset, because our inductive hypothesis $a_k \leq b_k$ implies $r_{a_k} \leq r_{b_k}$, and $\ell_{b_{k+1}} > r_{b_k}$ by construction. Thus, $\ell_{b_{k+1}} > r_{a_k}$, so we could have added the b_{k+1}^{th} interval to S_{alg}, a contradiction because when Algorithm 3 terminates, there were no more available intervals. Thus, $k = m$ and S_{alg} is optimal.

4.2 Balanced parentheses revisited

We next revisit the balanced parentheses problem from Section 2.3, Part III. Recall that our previous dynamic programming-based approach required $O(n^3)$ time. In this section we will solve a more complex variant of the problem with an $O(n)$-time greedy algorithm.

The variant we consider takes as input a length-n string S consisting of the characters '(' and ')', and outputs the minimum number of characters that must be flipped for S to be become balanced (cf. Section 2.3, Part III for the definition of balanced). Note that S was balanced to begin with iff the output to this new variant is 0, so this problem is strictly harder than our previous variant. As before, we assume n is even, as otherwise the problem is impossible.

We introduce one piece of helpful notation: for any string A consisting of '(' and ')' characters, let $\text{surplus}(A)$ denote the number of '(' characters minus the number of ')' characters in A.

To approach this problem, we begin by making two simple observations.

Lemma 3. *Let $B \in \{\text{'('}, \text{')'}\}^n$ be any balanced length-n parentheses string. For any $i \in [n]$, let $B[:i]$ denote the prefix of B consisting of the first i characters, and let $B[i:]$ denote the suffix consisting of the last $n - i + 1$ characters. Then,*

$$\text{surplus}(B[:i]) \geq 0, \ \text{surplus}(B[i:]) \leq 0, \ \textit{for all } i \in [n]. \tag{12}$$

Proof. To see the first condition in (12), if any prefix has more ')' characters than '(' characters, it is clear that some ')' characters must remain unmatched (as they can only be matched to preceding '(' characters), which would contradict B being balanced. Similarly, any suffix $B[i:]$ must have $\text{surplus}(B[i:]) \leq 0$, else some '(' character must go unmatched. □

We can now describe our greedy algorithm for solving our balanced parentheses variant.

Algorithm 4: BalParMinFlips(S)

```
1  Input: S ∈ {'(', ')'}^n
2  (flips, surplus) ← (0, 0)
3  for i ∈ [n] do
4      if S[i] == '(' then
5          surplus ← surplus + 1
6      end
7      else
8          if surplus > 0 then
9              surplus ← surplus − 1
10         end
11         else
12             flips ← flips + 1
13             surplus ← surplus + 1
14         end
15     end
16 end
17 return flips + (1/2) · surplus
```

We briefly explain Algorithm 4. Lines 3 to 16 take a pass over S, maintaining a counter of the current surplus of '(' characters compared to ')' characters. Whenever the surplus is about to become negative due to encountering a ')' when the surplus is 0 (i.e., the branch in Lines 11 to 14), the algorithm flips this ')' character to a '(' character immediately and continues. This flipping

rule is motivated by the first condition in (12): the surplus of any prefix must stay nonnegative, so the algorithm greedily flips characters to maintain this invariant.

We now prove optimality of Algorithm 4 using a greedy stays ahead argument. To describe the sense in which Algorithm 4 "stays ahead" we require one additional definition. Say that a length-i parentheses string F is *feasible* if it obeys the following rule:

$$\text{surplus}(F[:j]) \geq 0 \text{ for all } j \in [i]. \tag{13}$$

Note that any balanced parentheses string is feasible, by the first condition in (12).

We claim that for all $i \in [n]$, Algorithm 4 makes the fewest flips possible amongst the first i characters to ensure that $S[:i]$ becomes feasible (i.e., it stays ahead of the number of flips used by any other balancing algorithm). To ease notation, let alg_i denote the value of flips in Algorithm 4 after the first i loops of Lines 3 to 16, and let opt_i denote the fewest flips necessary (made by *any* algorithm) within $S[:i]$ to ensure it becomes feasible. Then our goal is to establish

$$\text{alg}_i \leq \text{opt}_i, \text{ for all } i \in [n]. \tag{14}$$

We prove (14) by induction. For the base case $i = 1$, the claim follows because Algorithm 4 flips the first character iff it is a '(', which is clearly necessary in any feasible string.

For the inductive step, assume that $\text{alg}_i \leq \text{opt}_i$. If the algorithm enters either of the branches on Lines 4 to 6 or Lines 8 to 10, then $\text{alg}_{i+1} = \text{alg}_i$. Moreover, clearly $\text{opt}_{i+1} \geq \text{opt}_i$, because any flips that make $S[:i+1]$ feasible must at least make $S[:i]$ feasible. Thus, $\text{alg}_{i+1} \leq \text{opt}_{i+1}$ as claimed.

It remains to discuss the case where Algorithm 4 enters the branch in Lines 11 to 14 on the $(i+1)^{\text{th}}$ loop. Suppose for contradiction that $\text{opt}_{i+1} < \text{alg}_{i+1} = \text{alg}_i + 1$. The key is to observe that

$$2 \cdot \text{alg}_i = -\text{surplus}(S[:i]), \tag{15}$$

because every flip made amongst the first i characters increases the surplus by 2, and the surplus of $S[:i]$ after all the flips made by Algorithm 4 in the first i loops is 0 (else we would have passed the check on Line 8). Thus, the original surplus of $S[:i]$ is precisely $-2 \cdot \text{alg}_i$. Now, if we use less than $\text{alg}_i + 1$ flips amongst $S[:i+1]$, the resulting surplus of the first $i+1$ characters is

$$\begin{aligned}\text{surplus}(S[:i+1]) + 2 \cdot \text{opt}_{i+1} &\leq \text{surplus}(S[:i+1]) + 2 \cdot \text{alg}_i \\ &= \text{surplus}(S[:i+1]) - \text{surplus}(S[:i]) = -1.\end{aligned}$$

The only inequality used the assumption $\text{opt}_{i+1} < \text{alg}_i + 1$, and the second line used (15) and the fact that $S[i+1] =$ ')'. This contradicts the feasibility of the new prefix of length $i+1$, because it does not have a nonnegative surplus, and thus $\text{opt}_{i+1} \geq \text{alg}_{i+1}$.

Finally, let us see why (14) shows Algorithm 4 solves the problem. Let $i^\star \in [n]$ be the last index that Algorithm 4 flipped from a ')' to a '('. Just before this point, the surplus is 0, so the final value of surplus when Algorithm 4 returns is the surplus of the original remaining suffix plus 2 (to account for the flipped $S[i^\star + 1]$), i.e., at termination,

$$\text{surplus} = \text{surplus}(S[i^\star :]) + 2, \text{ and flips} = \text{alg}_{i^\star}.$$

Our proof of (12) shows that any balancing algorithm must correct the prefix $S[:i^\star - 1]$ to become feasible, and flip $S[i^\star]$. Moreover, (12) also shows that any balancing algorithm must then correct $S[i^\star :]$ so that its surplus becomes nonpositive. We showed in (14) that the number of flips used by Algorithm 4, i.e., $\text{alg}_{i^\star}$, is optimal for the former. For the latter, every flip decreases the surplus by 2, and hence we need $\frac{1}{2} \cdot (\text{surplus}(S[i^\star :]) + 2)$ additional flips. Thus, the fewest flips needed is $\geq \text{flips} + \frac{1}{2} \cdot \text{surplus}$ for the values at termination, as achieved by Algorithm 4.

4.3 Minimum spanning tree

We conclude with a classical example combining ideas from both exchange and greedy stays ahead.

In the *minimum spanning tree* (MST) problem, we are given as input a connected undirected graph $G = (V, E, \mathbf{w})$. Our goal is to output a spanning tree $T \subseteq E$ with minimum total weight, i.e., a

Algorithm 5: MSTConceptual(G)

```
1 Input: G = (V, E, w), a connected undirected graph
2 Sort E in nondecreasing order by weight
3 T ← ∅
4 for e ∈ E do
5 |   if T ∪ {e} contains no cycle then
6 |   |   T ← T ∪ {e}
7 |   end
8 end
9 return T
```

tree subgraph of G achieving the smallest objective value defined by $f(T) := \sum_{e \in T} \mathbf{w}_e$. Recall from Section 4, Part I that a graph is a forest iff it contains no cycles, and a tree is a maximal forest in the sense that it has $n-1$ edges, and any graph with $\geq n$ edges must contain a cycle. We hence consider the following, very natural conceptual greedy algorithm for MST.

Algorithm 5 maintains a current set of edges T, and repeatedly tries to add edges to T, starting with the lowest-weight edge. If the edge currently under consideration can be added to T without creating a cycle, we greedily include it, and otherwise we move on to the next edge. Deferring implementation details to a later discussion, we claim that Algorithm 5 solves the MST problem correctly. This fact, attributed to [JBK56] so that Algorithm 5 is called Kruskal's algorithm, is perhaps quite surprising given the simplicity of our algorithm. We prove optimality of Algorithm 5 in this section, and show how it is representative of a more general phenomenon in Section **??**.

It is perhaps not even obvious that Algorithm 5 returns a tree. To see this, suppose for contradiction that at termination, T contains at least two distinct connected components. Then, there is some edge $e \in E$ joining these two components, else G would not be connected. When e is first encountered, $T \cup \{e\}$ contains no cycle, since T only grows in size over time. Hence, e would have been included by Line 6, a contradiction to the components being disconnected. What remains is to show that the resulting T is optimal for the MST problem.

In fact, we will prove a stronger claim. We claim that Algorithm 5 solves the *minimum spanning forest* problem correctly at every step: whenever $|T| = k$ in the execution of Algorithm 5 for any $k \in [n-1]$, the weight of T is optimal among any forest subgraph of G with k edges. This claim is highly reminiscent of the "greedy stays ahead" argument from Section 4.1. Applying this strengthened claim with $k \leftarrow n-1$ proves optimality of Algorithm 5 for MST.

To prove our stronger claim, we require a helper fact that underlies our exchange argument.

Lemma 4. *Let $G = (V, E, \mathbf{w})$ be an undirected graph, and let $F \subseteq E$, $F' \subseteq E$ be two forest subgraphs of G with $|F| < |F'|$. Then there is some $e \in F'$ such that $F \cup \{e\}$ remains a forest.*

Proof. Let $|F'| = n - c'$ and $|F| = n - c$ for some $1 \leq c' < c$. By Lemma 16, Part I, letting $\mathbf{w}_F \in \mathbb{R}^F$ denote the restriction of $\mathbf{w}$ to F, the subgraph $(V, F, \mathbf{w}_F)$ has c connected components. Similarly, the subgraph $(V, F', \mathbf{w}_{F'})$ has $c' < c$ connected components.

We claim that some edge $e \in F'$ joins two connected components in F. Indeed, if this were not the case, then every edge in F' lies in some connected component in F. The number of connected components in F' can thus only increase from F (each connected component in F either stays whole or is broken up into multiple pieces), i.e., $c' \geq c$, a contradiction.

So, some $e \in F'$ joins two connected components in F. Its inclusion cannot create a cycle in F, as otherwise there was already a path between the two connected components. □

Let us see how Lemma 4 helps us conclude our argument. For convenience, denote the edges added to T in Algorithm 5 by $\{e_1, e_2, \ldots, e_{n-1}\}$, where edges are added in sequence, i.e., e_1 is added first. Suppose for contradiction that after adding k edges to T, Algorithm 5 is suboptimal for the first time. This means that for any $j < k$, there exists no forest of size j with total weight $\leq \sum_{i \in [j]} \mathbf{w}_{e_i}$, but that there exists a forest $F' \subseteq E$ with $|F'| = k$ and $\sum_{e \in F'} \mathbf{w}_e < \sum_{i \in [k]} \mathbf{w}_{e_i}$.

We obtain a contradiction via Lemma 4. Denote the edges in F' by $\{e'_1, e'_2, \ldots, e'_k\}$. Let F be the forest consisting of $\{e_1, e_2, \ldots, e_{k-1}\}$. Then Lemma 4 states that there is some edge, without loss of generality e'_k, such that $F \cup \{e'_k\}$ is a forest. It must be the case that $\mathbf{w}_{e'_k} < \mathbf{w}_{e_k}$, as otherwise,

$$\sum_{e \in F} \mathbf{w}_e = \mathbf{w}_{e_k} + \sum_{i \in [k-1]} \mathbf{w}_{e_i} \leq \mathbf{w}_{e'_k} + \sum_{i \in [k-1]} \mathbf{w}_{e'_i} = \sum_{e' \in F} \mathbf{w}_{e'},$$

violating our earlier assumption $\sum_{e \in F'} \mathbf{w}_e < \sum_{i \in [k]} \mathbf{w}_{e_i}$. However, $\mathbf{w}_{e'_k} < \mathbf{w}_{e_k}$ also gives a contradiction, since Algorithm 5 should have picked e'_k instead of e_k in Line 6, as e'_k has strictly smaller weight than e_k (so it is encountered earlier) and also does not form a cycle when added to F.

Implementation details. We have shown that Algorithm 5 is optimal for the MST problem, but as written, it appears somewhat inefficient. However, letting $n := |V|$ and $m := |E|$, there is a simple implementation of Algorithm 5 that runs in $O(m \log(n))$ time, presented as Algorithm 6.

Algorithm 6: MST(G)

```
1  Input: G = (V, E, w), a connected undirected graph with n := |V| and m := |E|
2  Sort E in nondecreasing order by weight
3  C ← Array.Init(n)                          // Track connected components of all vertices.
4  for i ∈ [n] do
5  |  C[i] ← i                                // Initialize all connected components to size 1.
6  |  S_i ← Stack.Init()
7  |  S_i.Push(i)                            // S_i includes all vertices in the i^th connected component.
8  end
9  T ← ∅
10 for e = (u, v) ∈ E do
11 |  if C[u] ≠ C[v] then
12 |  |  T ← T ∪ e                           // Include edge e just as in Algorithm 5.
13 |  |  if |S_C[u]| ≥ |S_C[v]| then
   |  |  |                                   // Merge smaller connected component C[v] into bigger connected component C[u].
14 |  |  |  for k ∈ [|S_C[v]|] do
15 |  |  |  |  w ← S_C[v].Pop()
16 |  |  |  |  C[w] ← C[v]
17 |  |  |  |  S_C[u].Push(w)
18 |  |  |  end
19 |  |  end
20 |  |  else
   |  |  |                                   // Merge smaller connected component C[u] into bigger connected component C[v].
21 |  |  |  for k ∈ [|S_C[u]|] do
22 |  |  |  |  w ← S_C[u].Pop()
23 |  |  |  |  C[w] ← C[v]
24 |  |  |  |  S_C[v].Push(w)
25 |  |  |  end
26 |  |  end
27 |  end
28 end
29 return T
```

Algorithm 6 explicitly keeps track of the connected component $C[v]$ each vertex $v \in V$ belongs to, throughout the algorithm. It also creates lists (implemented as Stack instances) containing the members of each connected component. Originally, every vertex is in its own connected component. When an edge is discovered on Line 11 that would be included in the final tree by Algorithm 5, Algorithm 6 also includes the edge (because it joins two existing connected components). Algorithm 6 further updates the connected component information it maintains, by merging the smaller component into the larger one. Because it makes the exact same choices as Algorithm 5, we have proven Algorithm 6 also correctly produces an MST. We now discuss its runtime.

Sorting the edges in Line 2 takes $O(m \log(n))$ time, where we recalled $m \leq n^2$ so $\log(m) = O(\log(n))$. The initialization in Lines 10 to 28 takes $O(n)$ time. All steps of the loop from Line 10

to 28 run in $O(1)$ time except potentially the merging operations in Lines 13 to 19 and Lines 20 to 26. These operations pay $O(1)$ time per vertex that needs to change connected components.

Observe that every time a vertex changes connected components, it moves from a smaller component to a larger component, so the size of its connected component at least doubles with each move. This can only happen $O(\log(n))$ times, so the algorithm can only spend $O(\log(n))$ time merging any vertex in Lines 13 to 19 and Lines 20 to 26. The overall cost of merging operations is thus $O(n \log(n))$, and because we assume G is connected, $m \geq n-1$ so $O(n \log(n)) = O(m \log(n))$. Thus, the overall runtime of Algorithm 6 is $O(m \log(n))$ as claimed.

Improvements. The MST problem admits a variety of efficient algorithms, with various properties that may be preferable to Algorithm 6. The first such algorithm is due to Borůvka [Bor26]. Borůvka's algorithm has the added benefit of being easily parallelizable, as it tries to add as many cross-component edges as possible in each step, and each connected component can be handled by its own parallel thread. Another famous MST algorithm is Prim's algorithm [Pri57], which slowly grows a tree in each step by including the minimum-weight edge involving a non-tree vertex.

Using more sophisticated techniques, [KKT95] designed a randomized MST algorithm that runs in $O(m)$ time. Moreover, [Cha00] gave a deterministic MST algorithm that runs in $O(m\alpha(m,n))$ time, where $\alpha(m,n)$ is an extremely slow-growing function known as the *inverse Ackermann's function*. The current state-of-the-art is by [PR02], who surprisingly designed an MST algorithm achieving the optimal runtime for the problem, even if the asymptotic nature of that runtime is not yet known. All of these algorithms work in a restricted computational model where the only numerical operation allowed is comparing two edge weights to determine which is larger.

5 Stable matching

We conclude these notes with one of the most famous applications of greedy algorithms: stable matching. In the simplest variant of this problem, there are n applicants who we wish to pair with n job openings, such that every applicant is matched with exactly one job and every job is matched with exactly one applicant (this is called a *perfect matching*). The input to the problem is $2n$ different lists: each applicant submits a ranked order list of their preferences among the jobs, and each job opening submits a ranked order list of their preferences among the applicants. To simplify the problem, we assume that there are no ties in the preference orderings.

Our goal is to return a matching between jobs and applicants, that is *stable* in a precise sense. Essentially, we wish to prevent the following failure mode that can arise after our proposed matching is announced. Suppose we paired up applicant a with job α,[2] and applicant b with job β, but β preferred a to b, and a preferred β to α. Then, it is reasonable to expect that the job opening β and the applicant a may cut a backroom deal, where both a and β renege on the job-applicant pairs given by the matching, and internally pair themselves up instead.

Formally, a matching is *stable* iff for every potential backroom deal (a, β), where a is assigned the job $\alpha \neq \beta$ and β is assigned the applicant $b \neq a$, either a prefers its actual job assignment α over β, or β preferred its actual applicant assignment b over a. This implies that at least one of the parties a or β is disincentivized to make this deal. The stable matching problem asks to return a stable matching, when given preference orderings of all applicants and job openings as inputs.

This problem has many real-world applications, and indeed, its study was motivated by an algorithm that was already used by the National Resident Matching Program to place U.S. medical school graduates into residency training programs by the 1950s.[3] In Algorithm 7, we present a surprisingly simple algorithm by Gale and Shapley [GS62] that solves stable matching, which won a Nobel Prize in Economics in 2012. Before stating the Gale-Shapley algorithm, however, we wish to briefly motivate why simpler approaches to stable matching do not work.

[2] For notational clarity, we use English lowercase to denote applicants and Greek lowercase to denote jobs.

[3] The stable matching problem is sometimes presented in the context of marrying n men and women. We choose to present the applicant-job opening variant instead for several reasons. First, love is love, so the marriage example is somewhat dated. Second, we are not aware of any real-world applications of stable matching algorithms to marrying couples, contrary to the case of jobs. Finally, this presentation decision better highlights the asymmetry between the two parties, which is relevant when we analyze structural properties of the outputted matching.

Perhaps the simplest greedy approach to stable matching (inspired by Sections 2 and 3, which focused on the progress achieved by undoing inversions) is simply to swap any unstable pairs. Namely, the algorithm starts with an arbitrary perfect matching, and then repeatedly searches for unstable pairs (a, α), (b, β), fixing the instability by replacing these pairs with (a, β) and (b, α) in the matching. It is clear that if this procedure ever terminates, the matching is stable.

Unfortunately, it is possible for this naïve greedy algorithm to never terminate.[4] Consider a system with $n = 3$, where applicant a has the preference list (most-preferred ranked first) $\{\alpha, \gamma, \beta\}$, applicant b has the preference list $\{\gamma, \alpha, \beta\}$, and applicant c has the preference list $\{\alpha, \beta, \gamma\}$. Conversely, suppose that the job openings α, β, γ respectively have applicant preference rankings $\{b, a, c\}$, $\{c, a, b\}$, and $\{a, b, c\}$. If our initial matching is $\{(a, \alpha), (b, \beta), (c, \gamma)\}$, then undoing the unstable pairs (b, α), (b, γ), (a, γ), and (a, α) produces the following matchings in sequence:

$$\begin{aligned}\{(a,\alpha),(b,\beta),(c,\gamma)\} &\xrightarrow{(b,\alpha)} \{(b,\alpha),(a,\beta),(c,\gamma)\}\\ &\xrightarrow{(b,\gamma)} \{(c,\alpha),(a,\beta),(b,\gamma)\}\\ &\xrightarrow{(a,\gamma)} \{(c,\alpha),(b,\beta),(a,\gamma)\}\\ &\xrightarrow{(a,\alpha)} \{(a,\alpha),(b,\beta),(c,\gamma)\}.\end{aligned}$$

This example shows that swapping unstable matches can lead to a cycle, and therefore may fail to terminate with a stable matching.

The key idea that leads to breaking these cycles in the Gale-Shapley algorithm is the notion of a *temporary matching*, i.e., job offers. Initially, every applicant is unmatched, and we allow the job openings to make job offers. An applicant is allowed to renege on any temporary match (an offer they agreed to) and jump to any new offer they prefer. Similarly, if a job opening makes an offer to an applicant which is later reneged, they can make new offers to unmatched applicants.

We are now ready to state the Gale-Shapley algorithm.

Algorithm 7: StableMatching($\{A_a\}_{a\in[n]}, \{J_\alpha\}_{\alpha\in[n]}$)

1 **Input:** Applicant preference lists $\{A_a\}_{a\in[n]}$ which are permutations of $[n]$ ranking the n job openings (with most-preferred jobs first), and job opening preference lists $\{J_\alpha\}_{\alpha\in[n]}$ which are permutations of $[n]$ ranking the n applicants (with most-preferred applicants first)
2 $M \leftarrow \emptyset$ // Maintains current list of matched pairs.
3 $i_\alpha \leftarrow 1$ for all $\alpha \in [n]$ // Each job $\alpha \in [n]$ has pointer i_α to its favorite applicant who has not yet reneged.
4 **while** $\exists \alpha \in [n]$ with no $(a, \alpha) \in M$ **do**
5 $a \leftarrow J_\alpha[i_\alpha]$ // Unmatched job α makes an offer to its current favorite applicant $J_\alpha[i_\alpha]$
6 **if** $\exists \beta = A_a[j]$ such that $(a, \beta) \in M$ **and** $j > i$ where $\alpha = A_a[i]$ **then**
7 $M \leftarrow M \setminus \{(a, \beta)\} \cup \{(a, \alpha)\}$ // a reneges and temporarily accepts the new offer.
8 $i_\beta \leftarrow i_\beta + 1$ // a has rejected the old offer from β.
9 **end**
10 **else if** $\exists \beta = A_a[j]$ such that $(a, \beta) \in M$ **and** $j < i$ where $\alpha = A_a[i]$ **then**
11 $i_\alpha \leftarrow i_\alpha + 1$ // a has rejected the new offer from α.
12 **end**
13 **else**
14 $M \leftarrow M \cup \{(\alpha, a)\}$ // a temporarily accepts the new offer.
15 **end**
16 **end**
17 **return** M

There is a fair amount of notation in Algorithm 7, but it is quite straightforward to state what the algorithm is doing. The algorithm terminates whenever all openings have a matched applicant.

Before termination, in Line 4, the algorithm repeatedly asks any unmatched job opening α to make an offer to its favorite applicant $a = J_\alpha[i_\alpha]$ who has not yet reneged on, or rejected, an offer from

[4] Credit goes to Donald Knuth, by way of [Eri24], for this example.

α. If applicant a is unmatched, they temporarily accept the offer from α on Line 14. Otherwise, applicant a has already temporarily accepted an offer from some other job opening β. If a prefers β to its new offer from α, then it rejects the new offer and i_α increments due to the rejection (Line 11). Otherwise, a reneges on its old offer from β, and i_β increments (Lines 7 to 8).

Correctness. It is clear that Algorithm 7 yields a *perfect matching*, i.e., $|M| = n$ at termination. This is because otherwise, there is an unmatched applicant, so the loop must continue.

Next, we prove that the matching returned by Algorithm 7 is stable. Suppose for contradiction that (a, α) and (b, β) are both pairs in the final matching, but a prefers β to α, and β prefers a to b. The former fact implies that a never received an offer from β, because otherwise a would have ended up with a job at least as preferred as β. However, β could not make an offer to b before making an offer to a, as a appears earlier on β's preference list. This is a contradiction.

Runtime. We now need to address the elephant in the room: how do we know Algorithm 7 terminates at all? To prove that it does, we establish a notion of progress to prove that the loop from Line 4 to 16 can only run $O(n^2)$ times before the algorithm must terminate.

The key observation is that every run of the while loop, involving an unmatched job α and its current favorite applicant a, either ends with an offer being rejected (Lines 7 or 11), or a new matching being added to M (Line 14). Thus, in each loop, either one of the pointers $\{i_\alpha\}_{\alpha \in [n]}$ increments (due to a rejected offer), or the maintained matching M permanently grows in size. The former type of progress can only happen n^2 times, since each pointer lies in the range 1 to n. Similarly, the latter type of progress can only happen n times, since $|M| \leq n$. Additionally, note that no job opening can ever exhaust its entire preference list without the algorithm terminating, because once an opening has been rejected by every applicant, every applicant has accepted at least one job offer. Thus, we have shown that the while loop can only occur $n^2 + n = O(n^2)$ times.

To implement each run of the loop in $O(1)$ time, we maintain a Queue of all unmatched job openings, and remove an arbitrary such unmatched job opening from the Queue in each loop beginning on Line 4. Moreover, we can spend $O(n^2)$ time preprocessing the preference lists to also store inverse lookup lists (so that for a given applicant $a \in [n]$ and job opening $\alpha \in [n]$, we can look up the index $i \in [n]$ such that $J_\alpha[i] = a$ in $O(1)$ time, as required by Lines 6 and 10).

With these modifications, the overall stable matching algorithm runs in $O(n^2)$ time. This is a linear-time implementation in the size of the input, because it takes $O(n^2)$ time to specify all preference lists. Moreover, we remark that there are worst-case examples of preference lists and tiebreaking procedures to select unmatched jobs in Line 4 that can cause Algorithm 7 to take $\Omega(n^2)$ steps. The easiest such example is if all jobs agree on a preference ordering of applicants, and all applicants agree on a preference ordering of jobs. If all jobs take turns making offers to the globally best applicant in reverse order from the applicant's preference, and the applicant reneges on all offers except its best, this takes n offers to remove one applicant from the pool. Repeating for each applicant in turn can result in as many as $\frac{n(n+1)}{2} = \Omega(n^2)$ offers being made.

Structural properties. One interesting feature of the Gale-Shapley algorithm is that it works no matter how we select the unmatched job α to initiate a loop of Lines 4 to 16. This may seem like a bug: surely our algorithm details are underspecified? Incredibly, it turns out that Algorithm 7 always returns the same stable matching, regardless of how we implement Line 4!

There is a very fundamental reason why this is the case, which we summarize as follows.

Lemma 5. *Let M be the matching output by Algorithm 7, using any choice in the selection of an unmatched job opening in Line 4. Then the following hold.*

- *For every job opening $\alpha \in [n]$, suppose $(a, \alpha) \in M$. Then, if M' is any stable matching with respect to the same preference lists, if $(b, \alpha) \in M'$, then α prefers to a to b.*
- *For every applicant $a \in [n]$, suppose $(a, \alpha) \in M$. Then, if M' is any stable matching with respect to the same preference lists, if $(a, \beta) \in M'$, then a prefers to β to α.*

Lemma 5 has a clean interpretation. Say that an applicant a is *feasible* for a job opening α if $(a, \alpha) \in M$ for some stable matching M, and say that α is feasible for a similarly. Then Algorithm 7 gives each job their best feasible applicant, and gives each applicant their worst feasible job. This

also implies that there is only one possible matching that Algorithm 7 can output, because each job opening has a unique best feasible applicant.

Proof of Lemma 5. We first prove that every job obtains its best feasible applicant, regardless of how Algorithm 7 is executed. Suppose for the sake of contradiction that M produced by Algorithm 7 does not pair up some (a, α), where a is the best feasible applicant for α. Because α ends up with some feasible applicant, it made an offer to a at some point. Thus, a rejected α in favor of another job, β. Without loss of generality, suppose this is the first time in Algorithm 7 that a best feasible applicant for a job α rejects it for another job β.

Because a is feasible for α, there is some stable matching M' where (a, α) and (b, β) are paired, obtaining a contradiction. We claim that (a, β) is an unstable pair. We have seen that a prefers β to α, as this choice was made in Algorithm 7. Moreover, if β preferred b to a, then in executing Algorithm 7, for β to make an offer to a, b must have already rejected β. This is impossible, because b is feasible for β, and (a, α) was assumed to be the first instance of a best feasible rejection. Thus, β prefers a to b, concluding our proof that M' is in fact unstable if a ever rejects α.

Next, we prove the other part: that every applicant receives their worst feasible job. Suppose for the sake of contradiction that M produced by Algorithm 7 pairs up (a, α), where α is not the worst feasible job for a. Then there is some stable matching M' where a is paired up with β, that is disliked compared to α. Moreover, suppose that $(b, \alpha) \in M'$. Then we claim α prefers a to b: we already know from earlier that a is the best feasible applicant for α, and b is some feasible applicant. It follows that (a, α) is unstable in M', a contradiction. □

Further reading

For more on Sections 2.1 and 2.3, see Chapter 15.2, [CLRS22].

For more on Section 2.2, see Chapter 4.1, [Eri24].

For more on Section 3.1, see Chapter 13, [Rou22].

For more on Section 3.2, see Chapter 4.2, [KT05].

For more on Section 4.1, see Chapter 15.1, [CLRS22] or Chapter 4.2, [Eri24] or Chapter 4.1, [KT05].

For more on Section 4.3, see Chapter 21, [CLRS22] or Chapter 7, [Eri24] or Chapter 4.5, [KT05] or Chapter 15, [Rou22].

For more on Section 5, see Chapter 25.2, [CLRS22] or Chapter 4.5, [Eri24] or Chapter 1.1, [KT05].

References

[Bor26] Otakar Borůvka. O jistém problému minimálním. *Prce Mor. Přírodověd. Spol. V Brně III*, 3:37–58, 1926.

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition*. The MIT Press, 2022.

[Eri24] Jeff Erickson. *Algorithms*. 2024.

[GS62] David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–14, 1962.

[JBK56] Jr. Joseph B.K̃ruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KKT95] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[KT05] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 2005.

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002.

[Pri57] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[Rou22] Tim Roughgarden. *Algorithms Illuminated*. Soundlikeyourself Publishing, 2022.