NOI.PH Training: Algorithms 2

Greedy Algorithms

Carl Joshua Quines

Contents

1 Greedy or not?

We call an algorithm **greedy** if it makes the choice that looks like the best at the moment. It makes the locally best choice, hoping that it will lead to the best overall solution.

You're probably already familiar with a greedy algorithm. When giving change, cashiers often start with the largest possible denomination that they can give without going over, until all the change is given.

From the very description, we mostly apply greedy to **optimization problems**.[1] What is the minimum? What is the maximum? These are the kinds of problems that the greedy algorithm answers. But whether or not the greedy algorithm answers the *best* solution is a completely different thing entirely, as we'll now look at.

Note: By the time you're reading this document, I hope you're already well aware of the fact (from past problems and past training weeks) that **greedy algorithms are not always correct**! You need to have some sort of proof, a logical argument that it works.

1.1 Coin change

The following problem is called the **coin change problem**. Can it be solved with a greedy algorithm?

Example 1.1. You have a target amount of change c you need to give, and an unlimited number of coins of several denominations. What is the minimum number of coins you need to use to represent c?

Answer. Not always. Take the set of coins $\{4, 3, 1\}$. To make 6, a greedy algorithm would pick $4+1+1$ rather than the optimal $3+3$. So **the greedy algorithm doesn't always work here**. But sometimes, it does. For example, with our set of coins $\{10, 5, 1\}$, then the greedy algorithm works; here's an informal proof:

Proof. Suppose, instead of using a 5 when the greedy algorithm tells you to, you decide to do something else. Well, the only thing you *could* do is take a bunch of 1s. But if you have five 1s, **it's better to swap them out** with one 5. So if you're intent on not taking 5s while still making the *best* solution, you can only take at most four 1s. But if you only take at most four 1s, you can't make anything larger than 4.

Now suppose, instead of using a 10 when the greedy algorithm tells you to, you decide to do something else. Whatever you do, you have to only make it with 5s and 1s. From earlier, we can only use at most four 1s, and similarly, at most one 5. But if you do this, you can't make anything larger than 9. □

Exercise 1.2. Why doesn't the above proof work for $\{4, 3, 1\}$? (Unless otherwise stated, feel free to discuss all exercises on the Discord.)

Lesson. **The greedy algorithm doesn't always work.** Many times it only works for some specific cases, but not the general problem. So be cautious when you have a greedy algorithm that you don't have a proof for.

Remark. The kind of argument here is somewhat common. The structure looks like:

[1] We'll have examples of greedy ideas for non-optimization problems later on in the text, particular for math-style problems. But most of the time it will be for optimization problems.

1. Suppose we *don't* do what the greedy algorithm says at a certain step.
2. Then it would be better if we followed it anyway.

We'll discuss this technique more later; for now, let's look at more "greedy or not" examples.

1.2 Knapsack problem

Can the following problem be solved with the greedy algorithm?

Example 1.3. You have a knapsack that can carry a limited number of items. Given a set of items, each with a value, which items should you take so that you can carry them all in your knapsack while maximizing their total value?

Answer. The "obvious" greedy algorithm is *keep taking the item with the most value.* This works, and here's an informal proof:

Proof. Take whatever solution. If we can still pack items in our knapsack, it's better to take them, so assume our knapsack is full. Now, if there's an item outside the knapsack which has more value than an item inside the knapsack, **it's better to exchange it**, and eventually we end up with the greedy solution. □

The following problem is a slight modification to the above problem, and is called the **0–1 knapsack problem**. Can it be solved with the greedy algorithm?

Example 1.4. You have a knapsack that can carry a limited weight. Given a set of items, each with a weight and a value, which items should you take so that you can carry them all in your knapsack while maximizing their total value?

Answer. The "obvious" greedy algorithm is *keep taking the item with the most value per weight.* **This does not work.** Suppose your knapsack can handle a weight of 5 kg. Take the set of items with weights and values 1 kg, 3★, 2 kg, 5★, and 3 kg, 6★. The greedy algorithm will pick the first two items, but the best solution is the last two items.

The following problem is a slight modification to the above problem, and is called the **fractional knapsack problem**. Can it be solved with the greedy algorithm?

Example 1.5. You have a knapsack that can carry a limited weight. You're given a set of items, each with a weight and a value. **You can take fractions of items:** for example, if you take half of an item, it becomes half the weight and half the value. Which fractions of items should you take so that you can carry them all in your knapsack while maximizing their total value?

Answer. The "obvious" greedy algorithm is *keep taking the item with the most value per weight*, and this works! Here's an informal proof:

Proof. Suppose, instead of taking a whole item I when the greedy algorithm tells you to, you decide to take less of it. but then **it's better to swap out** less value per weight items with I, and we can always do this. □

Lesson. **Small changes to a problem can make or break the greedy algorithm.** If you see a problem and recall a similar problem that can be solved with the greedy algorithm, then it doesn't necessarily mean that it can be solved with greedy too.

1.3 Interval partitioning

The following problem is called the **bin packing problem**. Can it be solved with the greedy algorithm?

Example 1.6. You have a set of lectures, each with a given duration. All lectures must be scheduled within a specified start and end time (the same one for all lectures). You need to schedule lectures into classrooms such that a room is never used for two lectures at the same time. What's the minimum number of classrooms you need?

Answer. Here's a greedy algorithm called the *first-fit algorithm*. We sort the lectures from longest to shortest duration. For each lecture, we find the first classroom that can accommodate it, and schedule it to that classroom. If no such classroom exists, we use another classroom and schedule it for that classroom. **This does not work.**[2]

Exercise 1.7. Come up with a counterexample.

The following problem is called the **interval partitioning problem**. Can it be solved with the greedy algorithm?

Example 1.8. You have a set of lectures, each with a specified start and end time. You need to schedule lectures into classrooms such that no two lectures occur at the same time in the same room. What's the minimum number of classrooms you need?

Answer. Here's a greedy algorithm called *earliest start time*. We sort the lectures from earliest to latest start time. For each lecture, we find the first classroom that can accommodate it, and schedule it to that classroom. If no such classroom exists, we use another classroom and schedule it for that classroom. **This does work,** as we'll discuss in a later section.

Lesson. **Small changes to a problem can make or break the greedy algorithm**, as in the last lesson.

1.4 Interval scheduling

The following problem is called the **interval scheduling problem** or the activity selection problem. Can it be solved with the greedy algorithm?

Example 1.9. You have a set of lectures, each with a specified start and end time. You need to schedule lectures into **one classroom** such that no two lectures occur at the same time. What's the **maximum number of lectures** you can schedule?

Answer 1. Here's a greedy algorithm called *earliest start time*. We sort the lectures from earliest to latest start time. For each lecture, if it won't cause a conflict, we schedule it.

This does not work. Consider these lectures: $[0, 7], [1, 2], [3, 4], [5, 6]$. This greedy algorithm will take the first lecture and be done with it, but the optimal solution is to take the last three lectures. Let's try again.

Answer 2. Here's a greedy algorithm called *shortest duration*. We sort the lectures from shortest to longest duration. For each lecture, if it won't cause a conflict, we schedule it.

[2] But it does give a solution at most twice as bad as the best one.

This does not work. Consider these lectures: $[0, 4], [3, 6], [5, 9]$. This greedy algorithm will take the second lecture and be done, but the optimal solution is to take the first and last lectures. Let's try again.

Answer 3. Here's a greedy algorithm called *earliest end time*. We sort the lectures from earliest to latest end time. For each lecture, if it won't cause a conflict, we schedule it. It turns out that this *does* work, as we'll prove in a future section.

Lesson. **Small changes to a greedy algorithm can make or break it.** It turns out that finding the *right* greedy algorithm is sometimes a challenge on its own, as we'll discuss in the next section.

2 Finding greedy algorithms

Many greedy algorithms are natural. We discussed the algorithms for coin change and fractional knapsack, and both of these were pretty natural greedy algorithms.

On the other hand, some greedy algorithms seem like they come from thin air. How would you think of earliest start time for the interval partitioning problem, or earliest end time for the interval scheduling problem? In this section, we discuss two techniques that can be helpful for finding greedy algorithms that work.

2.1 Sorting

All of the greedy algorithms we've looked at so far involve **sorting** in *some sense*. Earliest start and end times both involve sorting. Fractional knapsack requires us to sort the items using value per weight. Even for coin change, we need to sort the denominations from most to least.

It's a good idea to **impose structure on your data**, and with sets of things the common way is to sort them.

Example 2.1. Consider a problem I posted as the description on my cover photo: https://www.facebook.com/photo.php?fbid=1553712074642167&set=a.507036645976387&type=3&permPage=1. There's really only one way to sort the data: sort the people by height. Then the natural greedy solution would be to place shorter people in the front, then taller people in the back, which works.

Example 2.2. Consider **Illegal Logging:** https://www.hackerrank.com/contests/noi-ph-2019-finals-2-mirror/challenges/illegal-logging. There are really only two ways to sort the data: by initial distance, or by velocity. And it doesn't really make sense to zap people by initial distance. So the natural greedy solution would be to zap people by decreasing velocity, which works.

2.2 Pick a subset, any subset

Suppose you have an optimization problem that involves choosing a subset. One general trick is to pick any subset. Then figure out when it would be better to swap an element from the subset with an element outside the subset. If the problem is suited to it, you'll end up with a greedy algorithm.

Example 2.3. One of the subproblems in **Baby Come Bak:** https://www.hackerrank.com/contests/noi-ph-2019-finals-1-mirror/challenges/come-bak-to-bakuna goes like this. You have a $2 \times n$ array of numbers, and an integer $k < n$. For each column, you pick one number to be red and one number to be blue, such that there are k red numbers in the top row. Which coloring minimizes the sum of the red numbers?

Well, suppose we color red any such set of numbers. Then suppose we have two numbers x_1 and y_1 in the top row, where x_1 is red and y_1 is blue. Let's name the corresponding numbers beneath them as x_2 and y_2. So x_2 must be blue and y_2 must be red.

When is it better to color y_1 red instead of x_1? It's better to swap if $x_1 + y_2$, the current choice of numbers, is larger than $x_2 + y_1$. But $x_1 + y_2 > x_2 + y_1$ rearranges to $x_1 - x_2 > y_1 - y_2$,

so we just find columns with small "top number minus bottom number".

So our greedy algorithm is this. Take the k columns with the smallest "top number minus bottom number". Then color the top number in these columns red, and color all the bottom number in all the other columns red. This solves the problem.

To me, this is the motivation for sorting by "top number minus bottom number". Of course, when I write up a formal proof for this, I can totally hide all of these details and make it look like magic. Even more magically, I could also observe that this is equivalent to Example 1.3!

Exercise 2.4. Construct an instance of the Example 1.3 problem that is equivalent to the $2 \times n$ array problem. "Equivalent" here means that a solution to one problem corresponds to a solution to the other problem in a "natural" way. You'll know it when you see it.

If you want a formal definition, you want to construct a bijection f carrying solutions of the first problem to the second problem, such that if X is a better solution than Y to the first problem, then $f(X)$ is a better solution than $f(Y)$ to the second problem, and vice versa.

As we'll see next section, this kind of trick naturally leads us to a method of proving our greedy algorithm is correct!

3 Proving greedy algorithms

There's already a lot of other resources when it comes to proving greedy algorithms work, so I won't emphasize this so much. Instead, I'll give a brief overview of some of the general methods. If you need more information, look at the other references outlined in the next section.

3.1 Exchange argument

In subsection 2.2, I mentioned that the method there naturally leads to a method of proving the greedy algorithm is correct. It's called the *exchange argument*, and the "pseudocode" looks like this:

1. Let X be the greedy solution and X' be the optimal solution. If they're the same, we're done.
2. Otherwise, they differ in some piece: X' contains some element that X doesn't, or X' contains two elements in a different order than X.
3. Transform X' by exchanging some piece of X' with some piece of X, using the previous step. **This is the step where you use the fact that X is chosen greedily.** This transformation shouldn't *worsen* X', so you end up with a different optimal solution X'.
4. If you keep doing this, eventually you'll end up with $X = X'$. So X has to be optimal too, and you're done.

Here's an example where we formalize the proof of Example 1.3.

Proof. Suppose that the greedy solution takes items with values $C_1 \geq C_2 \geq \cdots \geq C_n$, and that the optimal solution takes items with values $D_1 \geq D_2 \geq \cdots \geq D_n$. Let i be the smallest index such that $C_i \neq D_i$.

Because C_i was chosen greedily, we have $C_i \geq D_i$. So we can't worsen the optimal solution if we exchange D_i with C_i instead. Repeatedly doing this exchange with the optimal solution will yield the greedy solution, so we're done. □

Exercise 3.1. Write up a formal proof of the greedy algorithm in Example 2.3.

3.2 Greedy stays ahead

The **greedy stays ahead** argument, as its name suggests, proves that for each step of the algorithm, the greedy choice is at least as good as the optimal solution's choice, so it's at least as good as the optimal solution.

1. Let X be the greedy solution and X' be the optimal solution.
2. We pick a *measure* to formalize the *at least as good as the optimal solution's choice* part. Sometimes this is just the parameter we're maximizing, sometimes it's what we picked greedily.
3. Let $a_1, a_2, \ldots, a_n$ be the n measures of X, and let $b_1, b_2, \ldots, b_m$ be the m measures of X'. Then show that a_i is at least as good as b_i for all i. **This is the step where you use the fact that X is chosen greedily.** This is typically done by induction.
4. Use the fact that X is at least as good as X' in each measure to show that it's optimal. This is typically done by contradiction.

We'll use interval scheduling as an example.

Proof. Suppose the greedy solution picks tasks $X_1, X_2, \ldots, X_n$ and the optimal solution picks tasks $X'_1, X'_2, \ldots, X'_m$. The measure we'll be using is the end time. We'll show that for each i, the end time of X_i is earlier or the same as the end time of X'_i, and we show this by induction.

Since X_1 was chosen greedily, then the claim is true for $i = 1$. If the claim is true for i, then we know the end time of X_i is not later than the end time of X'_i. So any job that we can add to the current optimal solution, we can also add to the current greedy solution. **Since X_{i+1} is chosen greedily**, this proves the claim for $i + 1$.

So the end time of X_n is earlier or the same as the end time of X'_n. If X is not optimal, then $m > n$, and X'_{n+1} could have been added to X, contradicting the fact that X was chosen greedily. □

Exercise 3.2. Write up a formal proof of the greedy algorithm in Example 1.5.

3.3 Structural arguments

This argument isn't really a general template, but a catch-all category for "everything else". But one such similarity is that you're talking from a **global perspective**: you're looking at the entire structure at once. Hence why, in the literature, this is sometimes called the **structural argument**.

One technique is finding a lower bound or an upper bound, and then showing that the greedy algorithm achieves this bound.

Example 3.3. Consider the interval partitioning problem. Define the **depth** d of a set of lectures as the maximum number of lectures such that no two of them can be scheduled to the same classroom. Obviously, the number of classrooms we need is at least d. And it turns out that the greedy algorithm uses only d classrooms!

Exercise 3.4. Let n be the number of classrooms the greedy algorithm uses for interval partitioning, and let d be the depth. Show that $d \geq n$.

3.4 Extremal principle, but not really

If you have a math background, you're probably familiar with something called the *extremal principle* or the extreme principle. Here's a strong claim. **The extremal principle doesn't exist;** it's just a neat way to prove greedy algorithms.

Let's look at the classic example of the so-called extremal principle.

Example 3.5 (Putnam 1979)**.** Given n red points and n blue points in the plane, no three collinear, prove that we can draw n segments, each joining a red point to a blue point, such that no segments intersect.

There is a very natural greedy algorithm here. Take any configuration, then "uncross the crossings". That is, if segments AB and CD intersect, we replace them with segments AC and

BD instead, for some suitable labelling of the points. And if we try out a couple of examples on paper, this seems to always work.

Unfortunately, this algorithm doesn't always decrease the number of intersections after each step.

Exercise 3.6. Come up with an example of an uncrossing that *increases* the number of intersections.

But it *does* decrease the sum of the lengths of the segments (which is a structural thing, if you want to categorize it like that). Since the sum of the lengths of all the segments keeps decreasing, and there's only a finite number of possibilities it can have, it must eventually stop. And then it won't have any more intersections!

Now, if you want to write this neatly, you can do the "extremal principle" way:

Proof. Take the configuration of segments with the smallest sum of segment lengths (which exists since there are only a finite number of configurations). Suppose that two segments cross. Then we can uncross them and make the sum of segment lengths smaller, contradicting the way we picked our configuration. So no two segments cross and we're done. □

Yes, it's a very neat way to present the proof. But it's unnatural, and it hides the main idea of the proof anyway, which is to use a greedy algorithm. Of course, it *is* a neat way to present the proof.

4 Final notes

Here are some strategy notes on greedy algorithms, as well as other resources.

4.1 Smells like greedy

When does a problem feel like a problem that can be solved greedily? Some signs:

- **It's an optimization problem.** Not all optimization problems are solved greedily, and not all greedy algorithms solve optimization problems. But if you see this in conjunction with one of the signs below, you should take that as a hint.
- **It's similar to a classical problem.** Hopefully you've internalized that just because a problem is similar to a greedy problem, doesn't mean it can be solved greedily as well. But it *is* a clue that a similar approach may work. This is why it's especially important to **do lots of problems**, to build up the intuition to recognize when this happens.
- **Its bounds suggest a linearithmic solution.** A greedy algorithm, more often than not, runs in linearithmic ($\mathcal{O}(n \log n)$) time, which comes from the sorting. This in conjunction with the above two pretty strongly suggests a greedy approach for me.
- **It involves intervals.** Almost every problem I know of that involves intervals turned out to be a greedy problem. There's just something about intervals that makes them a common thing to do greedy on. When you see intervals, think about how to sort them or order them in a useful way.

Of course, just because a problem has the signs that it *can* be solved greedily, doesn't mean that a greedy solution exists. The right way to think about the above list is something that should pop out when you consider a problem, not something you should actively look for.

4.2 On proving greedy algorithms

As we've seen above, finding a completely rigorous proof can be hard, technical, and tedious. It takes even more time and effort if you want to write it formally. Thankfully, you don't have to do that. It's enough to just have a gist of the proof.

This doesn't mean you can rely on your *feelings* that much! As said above, changing just a small part of a problem can make or break a greedy solution.

If you have a greedy idea that you want to prove correct, then I suggest trying to apply the "exchange argument" or the "greedy stays ahead" arguments first. They work a lot of the time. If not, then you probably need a custom argument for the problem ("structural argument").

4.3 Proof by AC

If you can prove that a greedy algorithm for a problem works, then by all means program it and submit it! But in practice, and as you may have seen from the previous section, proving that a greedy algorithm works is often hard and tedious. Robin gives us some advice as to when to try a greedy algorithm:

- If the greedy algorithm does not take too long to code (around 5 to 10 minutes), then just code it and submit. If it works, that's proof by AC! Penalties aren't a thing for IOI-style contests, so if you get WA, it's just a few minutes wasted.[3]
- If the greedy algorithm will take longer to code, try **lots** of cases by hand and make sure your greedy algorithm passes them before coding.
- If it will take a very long time to code, then you probably want to try proving it formally, or at least knowing the gist of the proof, before you program.

There *are* nuances to this however. In particular, if you're trying a proof by AC:

- If you get a WA, you can't distinguish if it's because of the incorrect implementation or the incorrect algorithm. In this case, you can try to compare your code to a brute force, and try to figure out what goes wrong.[4]
- If you're joining an ICPC-style round and penalties are a thing, you may not want to attempt a proof by AC. You can decide whether the penalty is worth taking, or you can try to compare your code to a brute force to make sure.
- If you *do* decide to write a brute force, test millions of random small cases. Often, wrong greedy solutions break on relatively small cases anyway.

As Robin wrote last year, emphasis mine, "There are many optimization problems in recent contests, including the NOI.PH and the IOI, where the solution uses a greedy algorithm, but this greedy algorithm tends to be hard to spot and hard to prove. Indeed, missing a greedy solution can be a costly mistake, and can cause one to just lose out on a medal. Thus, **it may be a good use of your time to practice your intuition with greedy algorithms**."

On the other hand, here's something from Kevin: "I never really do proof by AC. I've only maybe done it once or twice in all contests I've joined. I always try to prove greedy solutions, even if informally. The thing is that I simply practiced proving greedy solutions enough so it's not anymore as hard for me to do them during contests. It also helps that you don't have to write it fully rigorously, so you can just draw a bunch of drawings to symbolize/illustrate your argument—you simply fill in the details in your head." So I guess one takeaway from this is to **do lots of problems**.

4.4 Other resources

And now the part where I link to other resources so I don't have to explain things myself!

- Slides from Kevin Wayne, explaining broadly the three different proof methods for greedy algorithms: http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf
- A Guide to Greedy Algorithms from a Stanford algorithms class, which explains in more detail greedy stays ahead and the exchange argument: https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/handouts/120%20Guide%20to%20Greedy%20Algorithms.pdf
- Two handouts from Cornell. The first explains greedy stays ahead: http://www.cs.cornell.edu/courses/cs482/2003su/handouts/greedy_ahead.pdf. The second explains the exchange argument: http://www.cs.cornell.edu/courses/cs482/2007sp/exchange.pdf.

[3] AC stands for All Clear, or full points on a problem; WA stands for Wrong Answer.
[4] This is general advice: it's much easier to debug a WA if you have a counterexample.

5 Problems

This week's problems follow the same policy as the ones in the `backtracking1` module. **Prioritize the problems in that module over the problems in this one,** as knowing complete search is a *way* more important skill.

5.1 Warmup problems

You are allowed to discuss these problems on the Discord.

W1 Chat Room: https://codeforces.com/problemset/problem/58/A

W2 Twins: https://codeforces.com/problemset/problem/160/A

W3 Jianzhi's Dinner Box: https://dunjudge.me/analysis/problems/1174/

W4 Random Teams: https://codeforces.com/problemset/problem/478/B

W5 Exams: https://codeforces.com/problemset/problem/479/C

W6 Ciel and Flowers: https://codeforces.com/problemset/problem/322/B

W7 Dragon of Loowater: UVa 11292

5.2 Non-coding problems

Submit these using the template in Google Classroom. No need to be too rigorous; as long as you're able to convince me, it's fine.

Solve at least [25★], and one problem worth at least [5★]. Problems in red are mandatory. Returning students are required to solve all of N6.

You are allowed to discuss the [2★] problems on the Discord.

N1 [2★] Exercise 3.4.

N2 [2★] Exercise 3.6.

N3 Consider each of the following problems and corresponding greedy algorithms. None of these greedy algorithms are correct. Construct a counterexample for each one.

(a) *Problem.* **Unit fraction decomposition:** A unit fraction is a fraction with numerator 1. Let r be a rational number. Find the minimum number of distinct unit fractions whose sum is r.

[2★] *Algorithm.* Keep subtracting the largest unit fraction less or equal to r that hasn't been used yet.

(b) *Problem.* **Minimum vertex cover:** In a graph, a *vertex cover* is a set of vertices such that each edge is incident to one of the vertices in the set. Find the minimum size of a vertex cover.

[2★] *Algorithm 1.* Choose the vertex with largest degree. Delete every edge incident to the vertex. Repeat until no edges remain.

[2★] *Algorithm 2.* Choose two adjacent vertices. Delete every edge incident to either vertex. Repeat until no edges remain.

(c) *Problem.* **Travelling salesman problem:** Given a weighted complete graph with n vertices, find a cycle of length n with minimum total weight.

[2★] *Algorithm 1.* Start on any vertex. From the current vertex, choose the nearest unvisited vertex as the next vertex. When all the vertices are visited, return to the starting vertex.

[3★] *Algorithm 2.* Go through all the edges from least to most weight. Select an edge if it does not cause a vertex to have degree three and does not form a cycle. When $n-1$ edges are selected, select the remaining edge to form the cycle.

(d) *Problem.* **Lean on Me:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/lean-on-me

[3★] *Algorithm 1.* From bottom to top, arrange the plywood from least to most weight.

[3★] *Algorithm 2.* From bottom to top, arrange the plywood from most to least strength.

[3★] *Algorithm 3.* From bottom to top, arrange the plywood from most to least strength minus weight.

N4 Suppose that the denominations $c_1 < c_2 < \cdots < c_n$ in the coin change satisfy c_i divides c_{i+1}, for each i.

(a) [5★] Prove that the greedy algorithm works.

(b) [3★] Is there a set of denominations that *don't* satisfy this divisibility condition, yet where the greedy algorithm still works?

N5 Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of lectures, where the lecture a_i runs for d_i minutes. You have one classroom to schedule all these lectures in some order; no two lectures can be scheduled at the same time. Let e_i be the end time of lecture a_i.

(a) [2★] Give an $\mathcal{O}(n \log n)$ greedy algorithm that schedules the lectures so as to minimize the average e_i.

(b) [7★] Prove that your algorithm gives the optimal solution.

N6 In this problem we lead up to Dirac's theorem, Ore's theorem, and Palmer's algorithm. Assume that all the graphs we're dealing with are simple (that is, have no edges joining vertices to themselves, or multiple edges between two vertices).

(a) [2★] Suppose that every vertex in a graph has degree at least δ. Let $v_1, v_2, \ldots, v_t$ be a longest path. Show that all the neighbors of v_t lie on the path. Explain why $t \geq \delta + 1$.

(b) [2★] Suppose that a graph has $2n$ vertices, and every vertex has degree at least n. Let $v_1, v_2, \ldots, v_t$ be a longest path. Show that all the neighbors of v_1 and v_t lie on the path. Explain why $t \geq n + 1$.

(c) [5★] In the previous setup, show that there is some k such that, v_1 and v_{k+1} are neighbors, and v_t and v_k are neighbors. Use this to form a cycle of length $t \geq n + 1$.

(d) [3★] In the previous setup, suppose that the cycle does not contain all the vertices in the graph. Then there exists some vertex v not in the cycle. Prove that v is a neighbor of one of the vertices in the cycle. Then derive a contradiction from $v_1, v_2, \ldots, v_t$ being a longest path.

This proves that all the vertices are in the cycle. So any graph with $2n$ vertices, where every vertex has degree at least n, has a cycle passing through all the vertices. This is known as Dirac's theorem.

(e) [7★] A similar proof to Dirac's theorem also proves the following theorem. Suppose that a graph with $n \geq 3$ vertices satisfies the property $\deg u + \deg v \geq n$ for any two distinct non-adjacent vertices u and v. Show that it has a cycle passing through all the vertices. This is known as Ore's theorem.

(f) [5★] Suppose that a graph satisfies the conditions in Ore's theorem. It turns out there's an efficient algorithm to find such a cycle. Label the vertices $0, 1, \ldots, n-1$ in some order, and take all labels modulo n. Suppose that i and $i+1$ are not neighbors. Show that there exists j such that i, $i+1$, j, $j+1$ are all distinct, i and j are neighbors, and $i+1$ and $j+1$ are neighbors.

(g) [5★] In the previous setup, after finding such j, reverse the vertices from $i+1$ to j, inclusive. In the new labelling, i and $i+1$ are now neighbors, and j and $j+1$ are neighbors too. Show that we can only repeat this "search and reverse" step at most n times, and at the end, we have a cycle passing through all the vertices.

Since the "search and reverse" step can be done in $\mathcal{O}(n)$, the whole algorithm is $\mathcal{O}(n^2)$. This is known as Palmer's algorithm.

It is also worth noting that any graph satisfying the conditions in Ore's theorem have $\Omega(n^2)$ edges (can you see why?), so the algorithm actually runs in time linear in the input size.

5.3 Coding problems

Submit problem **CX** in the file `algo2-CX.cpp` or `algo2-CX.java`.

First timers, solve at least [25★]. Try to solve as many red problems as you can. The only red graph theory problem is C1; feel free to discuss this problem if you need help. If you finish `backtracking1` early, aim for [40★].

Returning students, solve at least [50★]. Problems in red worth less than [8★] are mandatory. You are required to solve at least two problems worth at least [9★]. If you finish `backtracking1` early, aim for [70★].

You are allowed to discuss the [2★] problems on the Discord.

C1 [2★] **Hamiltonish Path:** https://atcoder.jp/contests/agc013/tasks/agc013_b (You want to read N6 first.)

C2 [3★] **Frickin' Heck:** Problem L in http://penoy.admu.edu.ph/~acmicpc/wp-content/uploads/ACM-ICPC-Manila-2017-Problem-Set.pdf

C3 [5★] **Art Exhibition:** Problem 2 in https://www.ioi-jp.org/joi/2017/2018-ho/2018-ho-en.pdf

C4 [7★] **Hiring:** http://www.ioi2009.org/GetResource?id=1270

C5 [9★] (Adapted from IMO 2003.) You are given a set A of 101 distinct integers between 1 and 10^6, inclusive. Output 100 integers $t_1, t_2, \ldots, t_{100}$, each between 1 and 10^6, inclusive, such that the sets

$$A_j = \{x + t_j \mid x \in A\}, \qquad j = 1, 2, \ldots, 100$$

are pairwise disjoint. Your program should run within "the usual time limits".

C6 [15★] **Teleporters:** https://ioinformatics.org/files/ioi2008problem6.pdf

C7 [15★] **Backup:** Problem 2 in http://apio-olympiad.org/2007/apio-en.pdf

---

S1 [2★] **Hackerland Radio Transmitters:** https://www.hackerrank.com/contests/master/challenges/hackerland-radio-transmitters

S2 [2★] **The Bus Driver Problem:** UVa 11389

S3 [2★] Subtask 4 of **Skyscaping:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/skyscaping

S4 [2★] **Add All:** UVa 10954

S5 [3★] **Minimal coverage:** UVa 10020

S6 [3★] **The number on the board:** https://codeforces.com/problemset/problem/835/B

S7 [3★] **Queue:** https://codeforces.com/problemset/problem/545/D

S8 [3★] **Woodcutters:** https://codeforces.com/problemset/problem/545/C

S9 [5★] **Outside the Battlements:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/ebf-4

S10 [5★] Subtask 3 of **Guardians of the Lunatics Vol. 2:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/guardians-lunatics-2

S11 [5★] **A Match Making Problem:** UVa 12210

S12 [5★] **Bridge:** UVa 10037

S13 [5★] **Squares and not squares:** https://codeforces.com/problemset/problem/898/E

S14 [7★] **Dynamic Frog:** UVa 11157

S15 [7★] **Coin Collector:** UVa 11264

S16 [7★] **Make palindrome:** https://codeforces.com/problemset/problem/600/C

S17 [7★] **Mandragora Forest:** https://www.hackerrank.com/contests/master/challenges/mandragora

S18 [7★] **The Trip, 2007:** UVa 11100

S19 [9★] **Clementi:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/singapore-4 (Make sure to use `unsigned long long`.)

S20 [9★] **Papers, Please:** https://www.hackerrank.com/contests/noi-ph-2018-preselection/challenges/papers-please

S21 [9★] **Alarm Clock:** https://codeforces.com/problemset/problem/898/D

S22 [9★] **Hamiltonian Cycle:** UVa 775 (You want to read N6 first.)

S23 [12★] **Sereja and the Arrangement of Numbers:** https://codeforces.com/problemset/problem/367/C

S24 [12★] **New Year and Rainbow Roads:** https://codeforces.com/problemset/problem/908/F

S25 [15★] **Lean on Me:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/lean-on-me