1 Greedy Algorithm

A greedy algorithm is an algorithm where you want to build up some solution one step at a time. At every step, you come up with some notion of what the *best* choice you can make is, and make that choice; in other words, at every step, you make the *best available choice* until you can no longer make any choices.

A few things to keep in mind regarding greedy algorithms:

- They are very simple (almost trivial) and clean to write. *When they work*, they tend to be very good algorithms.
- It's not uncommon to write a greedy algorithm that seems to work, but actually fails. Thus, proving correctness is very important.

1.1 Problem: Making Change

Given some amount of money (in US dollars), you want to make the exact change for it using as few bills and coins as possible. For example, if we wanted to make change for a single dollar, you would use a single dollar bill instead of 100 pennies.

1.1.1 Greedy Algorithm

The algorithm is to repeatedly add the biggest denomiation still available.

1.1.2 Example: Making Change

What is the most optimal way to make change for \$12.73? Here, we define optimal as using the least number of bills and coins to make exact chamge.

Suppose we wanted to make change for \$12.73.

- We can begin with a change for \$10 bill, so we now need to make change for \$2.73.
- We can next use two \$1 bills, so we now need to make change for \$0.73.
- Next, we can use two quarters, so we now need to make change for \$0.23.
- Next, we can use two dimes, so we now need to make change for \$0.03.
- Finally, we can use three pennies.

At the end, we only needed 10 different bills and coins to make change.

1.1.3 Result

Proposition. *For standard US currency denomiations, this algorithm is always optimal.*

Proof. This proof is omitted. □

1.1.4 Non-Application to Other Currencies

This property does not necessarily hold for other currencies.

For example, suppose we have a country with currency W, where the denominations are $7W$, $5W$, and $1W$. What if we try to make change for $10W$?

- We first use $7W$, leaving us with $3W$ left to make change.
- Then, we use three $1W$ to cover the rest of the change needed.

This leaves us with 4 different bills and coins. However, the optimal solution is actually to use two $5W$ instead.

1.2 Problem: Interval Scheduling

You are trying to figure out your classes for the next quarter, and you are at a school where classes have weird times (e.g. 3:15 PM to 3:27 PM). You're not allowed to pick two classes that overlap. However, beyond this, you are a masochist and so you want to schedule as many classes as you possibly can; you don't care what classes you want to take, just that you can take them without having any overlaps.

More formally, given a set S of intervals $[x_i, y_i] = I_i$, you want to find a subset $T \subseteq S$ such that

1. No two intervals in T overlap.
2. Subject to condition 1, $|T|$ should be as large as possible.

For example, if we have the following intervals:

```
|----------------------|
    |-----| |------------|
  |------| |-------| |------|
```

It's not hard to see that the answer would be the three bottom intervals.

1.2.1 Greedy Algorithm Idea

We want to build this schedule one interval at a time. Now, we want to figure out which intervals we should be adding. Some ideas include:

- Shortest intervals, because longer intervals are more expensive to add.
- Fewest overlaps.
- Ends soonest.
- Starts soonest.

Picking the shortest interval doesn't work; consider this counterexample, where the two longest intervals represent the best possible solution:

```
|-----------------------------| |-----------------------------------|
                            |-----|
```

Picking the intervals with fewest overlaps doesn't work; consider this counterexample, where picking the interval with the fewest overlaps locks ourselves from the best possible schedule:

```
|------| |------| |------| |------|
    |-------| |------| |-------|
    |-------|          |-------|
    |-------|          |-------|
    |-------|          |-------|
```

Picking the intervals that start the soonest also doesn't work; consider this counterexample, where picking the interval that starts the soonest locks ourselves from the best possible answer:

```
|--------------------------------------------|
    |-----|                     |----|
```

However, picking the interval that ends the soonest works! We define the interval that ends the soonest to be the interval where the endpoint is minimal. In the example below, (a) ends the soonest because its endpoint is minimal (compared to (b)'s endpoint):

```
    (a)
|------|
    |------|
       (b)
```

Although not a proof, picking the intervals that ends the soonest in the above counterexamples gives us the best possible schedule.

1.2.2 Intuition

What does it mean for T to have no overlapping intervals? Well, if T has no overlaps, then we can sort T in some order; that is,

$$T = \{J_1, J_2, \dots, J_k\}$$

such that

$$J_1 < J_2 < \dots < J_k$$

When we say that J_1 comes before J_2, we essentially are saying that if $J_1 = [x_1, y_1]$ and $J_2 = [x_2, y_2]$, then $y_1 < x_2$. But, this is the only requirement on the interval J_1.

1.2.3 Formal Proof

We now want to formally prove that this is the case.

Proof. Suppose the greedy algorithm produces the set T such that

$$J_1 < J_2 < \dots < J_m \qquad J_i = [x_i, y_i]$$

and each of $J_i \in T$. We will now prove, by induction on k, that <u>any</u> set of intervals

$$I_1 < I_2 < \dots \qquad I_i = [x'_i, y'_i]$$

has I_k ending no sooner than J_k.

- <u>Base Case:</u> For $k = 1$, we know that J_1 ends the earliest and so

$$y_1 \le y'_1$$

- <u>Inductive Step:</u> Suppose that this is true for k; that is, $y_k \le y'_k$. So, J_{k+1} has the smallest possible y for any interval with $x > y_k$. We know that $I_{k+1} > I_k$, which means that $x'_{k+1} > y'_k$. By the inductive hypothesis, we know that

$$x'_{k+1} > y'_k \ge y_k$$

  This all implies that

$$y_{k+1} \le y'_{k+1}$$

And this completes the proof. □

1.2.4 Greedy Algorithm

```
BestInterval(S):
    let T = {}
    while S not empty:
        let J = S with minimum end time
        add J to T
        remove all intervals overlapping J from S
    return T
```

The `while`-loop takes $\mathcal{O}(n)$ time, and finding the minimum element and removing all intervals overlapping J takes $\mathcal{O}(n)$ time, so the runtime is approximately $\mathcal{O}(n^2)$ time.

To optimize the algorithm, we can sort S by the end time.

```
BestInterval(S):
    let T = {}
    Sort S by end time
    yMax = -inf
    For I in S in order:
        If start(I) > yMax:
            add I to T
            yMax = end(I)
    return T
```

In our optimized algorithm, it takes $\mathcal{O}(n\log(n))$ time to sort and then $\mathcal{O}(n)$ time to iterate over the intervals. Therefore, the optimized runtime is approximately $\mathcal{O}(n\log(n))$ time.