1 Greedy Algorithm

We continue our discussion of greedy algorithms. Recall that a greedy algorithm does the following:

1. Build solution one step at a time.
2. For each step, make the "best" possible choice.
3. Continue doing this until there are no more choices.

1.1 Exchange Argument: Proving Correctness of Greedy Algorithms

The exchange argument is a generic way[1] to prove correctness of a greedy algorithm. The idea for the exchange argument is as follows:

- Start from an arbitrary solution A.
- Slowly turn A into the greedy solution G, making it only better each step.
- Conclude that G is just as good as A, if not better. From there, you can conclude that since A is arbitrary, then G must be optimal.

1.1.1 Layout

Construct a sequence of solutions

$$A = A_0 \leq A_1 \leq A_2 \leq \cdots \leq A_n \leq G$$

where A_t agrees with the first t greedy choices. Then, given any A_t that agrees with the first t greedy choices, there exists an A_{t+1} that agrees with the first $t+1$ greedy choices and A_{t+1} is no worse than A_t, i.e. $A_{t+1} \geq A_t$.

Define an arbitrary A, we can say that $A = A_0$ (you made no choices so there's nothing to be consistent). From the layout above, we can construct an A_1. By induction, we can continue until we have an A_n, where n is the number of choices that you had to make for the greedy algorithm. Finally, we need to show that $A_n \leq G$.

So, really, the goal is to find a sequence of solutions

$$A = A_0, A_1, \ldots, A_n = G$$

such that

- $A_i \leq A_{i+1}$ (i.e. A_{i+1} is just as good as A_i, if not better).
- A_i agrees with $D_1, D_2, \ldots, D_i$, where D is the decision that the greedy algorithm made.

1.1.2 Example: Interval Scheduling Problem

We will show that the greedy solution from the interval scheduling problem is, indeed, optimal.

Proof. Let the greedy solution be defined by

$$G = J_1, J_2, \ldots, J_n$$

[1] Of course, it may not always work and it may not be the best.

such that

$$A_t = \underbrace{J_1 < J_2, < \cdots < J_t}_{\substack{A_t \text{ is some solution that agrees with the first } t \text{ decisions made by the greedy algorithm.}}} < \overbrace{I_{t+1} < I_{t+2} < \cdots < I_m}^{\substack{\text{After that, these can be any arbitrary intervals as long they don't overlap.}}}$$

We want to show that, given that the above is some valid set of intervals, we can find an A_{t+1} that is no worse than the above (i.e. no fewer intervals that A_t) but also includes the first $t+1$ intervals of the greedy algorithm. There are two cases to consider:

1. If $m = t$ (if A_t only has those t intervals), then there's a clear way to improve it - by adding the next interval. So,
$$A_t < A_{t+1} = J_1 < J_2 < \cdots < J_{t+1}$$

2. If $m > t$, then we can change the current solution to the solution which includes J_{t+1} by replacing I_{t+1} by J_{t+1}. That is,
$$A_{t+1} = J_1 < J_2 < \cdots < J_{t+1} < I_{t+2} < \cdots < I_m$$
It's clear that $A_{t+1} \geq A_t$. It's also clear that it agrees with the first $t+1$ decisions of the greedy algorithm. We just need to check whether A_{t+1} is valid (i.e. if we add A_{t+1}, then will the intervals overlap?). One thing we need to verify is whether the intervals are disjoint. Now, because G is valid, it's clear that
$$J_1 < J_2 < \cdots < J_{t+1}$$
and because A_t is valid, then
$$I_{t+2} < I_{t+3} < \cdots < I_m$$
So, we need to show that $J_{t+1} < I_{t+2}$. Recall that J_{t+1} has the smallest max among all intervals greater than J_t. Since $I_{t+1} > J_t$, this means that
$$\max(J_{t+1}) \leq \max(I_{t+1})$$
But because the I's don't overlap, that says that
$$\min(I_{t+2}) > \max(I_{t+1}) \geq \max(J_{t+1})$$
But, this shows that $I_{t+2} > J_{t+1}$, so we are done.

This concludes the proof. □

1.1.3 Summary

So, the end idea is that we started with an arbitrary sequence

$$A_0 = I_1 < I_2 < \cdots < I_m$$

such that none of the decisions agree with the greedy solution, we can transform them so that the first interval agree

$$A_1 = J_1 < I_2 < \cdots < I_m$$

and then the second interval

$$A_2 = J_1 < J_2 < \cdots < J_m$$

If we keep going with this, eventually we'll end up with

$$A_m = J_1 < J_2 < \cdots < J_m$$

Now, expanding upon this, we have

$$A_{m+1} = J_1 < J_2 < \cdots < J_m < J_{m+1}$$

and then eventually we'll reach the desired conclusion

$$A_n = J_1 < J_2 < \cdots < \cdots < J_n$$

In particular, $n \geq m$ and so the greedy solution is just as good as any other solution.

1.2 Problem: Optimal Caching

To give some background, we discuss the memory hierarchy in a very simplified form.

- Your computer has a CPU and a disk, which stores your memory.
- On old computers, your disk was often a spinning platter of metal. To read a random entry, you need to spin this metal until the head points to the appropriate location.
- Your disk operates on the order of milliseconds; however, your CPU operates on the order of nanoseconds.
- To solve this, you also have a cache, which is often stored on the same chip as the CPU.
- If we need to access some memory many times, instead of storing it on the disk, we can store it on the cache. If the memory is in the cache, it is very easy to look up since it is right there on the CPU on the chip.
- The cache is relatively small; we assume that the cache can store k words.

An unrealistic assumption that we'll make is that we know what memory accesses the program is going to make ahead of time. Consider the following example below, where $k = 2$ and we make another assumption that whenever we want to operate on some point in memory, we always load it in cache instead of in the disk.

```
time           ------------------------------------------------
memory access  a  b  a  b  c  a  d  e  c  b  c  a  c

cache 1        a  -  -  -  -  -  -  -  c  -  -  -  -
cache 2           b  -  -  c  -  d  e  -  b  -  a  -
```

This particular memory/cache schedule satisfies the sequence of memory accesses that are made. At every time when some location in memory needs to be accessed, it is already in one of the two locations in the cache. As for how expensive this is, suppose that the cost of the schedule is the number of times we need to load the new memory into cache. So, in our example above, this would be **8** cache misses (8 times when we needed to read something that wasn't in the cache and thus needed to be loaded from disk).

This leads us to the optimal caching problem. Given k and a sequence of memory accesses, find a consistent cache schedule[2] such that the number of loads from disk is minimized.

So, with the example above, the memory/cache schedule needed to load from disk 8 times. Can we do it in fewer times?

[2]Schedule of what memory location is in cache at what time in such a way that when we need to access that memory location, it is already in cache (maybe it was just loaded there, but it's there)

1.2.1 Observation

We should only load new memory on a cache miss. In other words, if we need to load some memory that isn't in cache, then we will. Now, the question becomes: when we load something new in memory, what do we throw out? i.e. if cache 1 has a and cache 2 has b, and we needed to load c, what would we throw out of cache?

- Least recently used: throw out the memory that was least recently used.
- Furthest in the future: since we have the sequence of memory accesses (including the future ones), we can see, in memory, how long it will take before that particular location is needed again, and the one that is furthest in the future is the one that we throw out.
- Least frequently used: throw out the memory that isn't being used frequently.
- Least frequently used in the future.

Here, *furthest in the future* works.

1.2.2 Proof that Furthest in the Future Works

Proof. We make use of the exchange argument. Here, we say that A_t agrees with G, the greedy solution, for the first t timesteps. That is, up to time t, both A_t and G do exactly the same thing; after time t, they do something else.

```
timestep ---------------------------- t -------------
A        xxxxxxxxxxxxxxxxxxxxxxxxxxxxx aaaaaaaaaaaaa
G        xxxxxxxxxxxxxxxxxxxxxxxxxxxxx ggggggggggggg
```

We want to show how to get from A_t to A_{t+1}. In other words, if we can find something that agrees with G for t timesteps, we can turn it into something that agrees G for $t+1$ timesteps and is no worse (no more extra memory loads than we had before). To do this, we need to show a bunch of different cases.

- Case 1: No cache misses at time $t+1$. In this case, G does nothing.
  - Case 1a: A_t also does nothing. In this case, $A_{t+1} = A_t$.
  - Case 1b: A_t does replace something. In this case, to get A_{t+1}, we put off those replacements by one timestep.

    ```
    timestep ------------------ t --- t+1 --- t+2
    A                           x     a
                                y     b
                                z     c
    G                           x               a
                                y               b
                                z               c
    ```

    So, if A_t replaces x, y, and z by a, b, and c, then because there wasn't a cache misses, then the memory access being asked for at timestep $t+1$ is none of a, b, or c, and so in particular we can put off replacing these by a, b, and c until timestep $t+2$.
- Case 2: Cache miss at time $t+1$. Here, we make a reduction that A_t only replaces the missed element. In other words, say we had to load up w but the cache already has a, b, and c; then, we can replace a with w and keep b and c.
  - Case 2a: Load into same location as G. Then, $A_{t+1} = A_t$ since we already agree with the greedy solution.

– Case 2b: Load into a different location than G. In other words, suppose at time $t+1$ we needed to load w, but the cache for both A_t and G had x, y, and z. Then, maybe the cache will replace y with w while the greedy will replace x with w; that is:

```
                                    (w)
timestep ------------------ t --- t+1
A                           x     a
                            y     w
                            z     c
G                           x     w
                            y     b
                            z     c
```

In this case, we need to consider several subcases.

* Case 2bi: A_t throws out x before using it. For example,

```
                                          (w)      (p)
timestep ------------------ t --- t+1 --- t+2
A                           x     -        p
                            y     w        -
                            z     -        -
A_{t+1}                     x     w
                            y     -        p
                            z     -
```

In this case, one way to think about it is that A_t throws out y and then x. Instead, we can throw out x and then y.

* Case 2bii: Need x first. Then, x is furthest in the future, so there must be some call y also early.

This concludes the proof. □