CS 4349.003.19F Lecture 10—September 23, 2019

Main topics for #lecture include #greedy_algorithms including #example/class_scheduling.

Storing Files on Tape

- Suppose we are asked to store n files on a magnetic tape. This concept may seem archaic, but they are still used for storing massive amounts of data in supercomputing facilities.
- Files on tape are stored in order. In order to read a file, you have to move your tape head past all the files stored before it. Let L[1 .. n] be an array listed the length of the files so file i has length L[i]. If the files are stored in order from 1 to n, then the cost of accessing the kth file is

$$cost(k) = \sum_{i=1}^{k} L[i].$$

- Now, suppose we are equally likely to access any of the n files. We can (should) define the *expected cost* of searching for a random file as

$$\mathrm{E}[cost] = \sum_{k=1}^{n} \frac{cost(k)}{n} = \frac{1}{n} \sum_{k=1}^{n} \sum_{i=1}^{k} L[i].$$

- If we reorder the files, then we change the expected cost of a random access. Some files become more expensive to read while others become cheaper.
- We can model reordering the files as picking a permutation pi where pi(i) tells you the index in L of the ith file on the tape. The expected cost of the permutation pi is

$$\mathrm{E}[cost(\pi)] = \frac{1}{n} \sum_{k=1}^{n} \sum_{i=1}^{k} L[\pi(i)].$$

- Now let's design an algorithm for the following problem: Given the array L[1 .. n] of file sizes, find a permutation pi that *minimizes* the expected cost of pi.
- The previous few lectures suggest we should begin with a backtracking approach. Let's try to make one decision about where a file is placed and then recursively place the rest of the files.
- And I'd argue most natural decision to make is what do we use for our first file. A backtracking algorithm would iteratively try each file as the first while recursively placing the remaining files.
- And this would give us a reasonable algorithm! Suppose we know file pi(1). Then:
  - 1 / n \sum_{k = 1}^n \sum_{i = 1}^k L[pi(i)]
  - = 1 / n (n * L[pi(1)] + \sum_{k = 2}^n \sum_{i = 2}^k L[pi(i)])
  - = L[pi(1)] + (n - 1) / n * 1 / (n - 1) \sum_{k = 2}^n \sum_{i = 2}^k L[pi(i)]

so, having fixed pi(1), the expected cost is minimized if we minimizing the expected cost for storing files 2 through n.

- There's a pretty big issue with this approach, though. Given we've guessed some number of files to go near the beginning of the tape, we now need to tell the Recursion Fairy what subset of files remain to be place. There are 2^n subsets, meaning 2^n subproblems. Even using dynamic programming, our algorithm is going to be very very slow.
- Fortunately, this problem just so happens to have a very nice structure that lets us skip almost all of the recursive calls. It turns out there is a best choice we can make for the first file *before* we do recursion. So we'll remove the loop and instead make the choice and do a single recursive call to deal with the consequences.
- Lemma: Let x be a file minimizing L[x]. Some optimal placement of files has pi(1) = x.
- In other words, we can place the shortest file first, and then recursively place the remaining files.
- Proof:
  - Let i be such that x = pi(i).
  - If i = 1, the claim is true, otherwise…
  - Let's move x to the first position on the tape and push everything else back.
  - Any file pi(k) for 1 ≤ k < i just had its access cost increase by L[x].
  - However, the cost of accessing x has now decreased by L[k] for all 1 ≤ k < i.
  - In other words the expected cost of pi just changed by
    - 1 / n (sum_{k = 1}^{i - 1} L[x] - sum_{k = 1}^{i - 1} L[k])
    - = 1 / n (sum_{k = 1}^{i - 1} (L[x] - L[k])
    - ≤ 1 / n (sum_{k = 1}^{i - 1} 0
    - = 0
  - The new permutation is no worse, so it must also be optimal!
- We can implement our recursive strategy by first sorting in increasing order of length and placing the files in the sorted order. That's only O(n log n) time in addition to however long it actually takes to write the files to the tape.
- What we just saw was an example of a *greedy algorithm*. The way I like to think about it is that we're still using a recursive strategy, but instead of looping over all the choices for our first decision, we made a best choice right away *before* doing recursion. It's like backtracking without having to track back.
- The proof we used is an example of an *exchange argument*. We took an arbitrary optimal solution and exchanged some aspect of it with another so it was consistent with the first choice we wanted to make. The new solution remained optimal, so we know our first choice is safe.
- We'll see more greedy algorithms and exchange arguments over the next couple days.
- In fact, let's start by generalizing our previous idea just a bit.
- Suppose now that some files are expected to be accessed more often than others.

- We're given an array F[1 .. n] of *access frequencies* for each file; file i will be accessed exactly F[i] times over the lifetime of the tape.
- Now the *total cost* of accessing all the files on the tape is

$$\Sigma cost(\pi) = \sum_{k=1}^{n} \left(F[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)] \right) = \sum_{k=1}^{n} \sum_{i=1}^{k} \left(F[\pi(k)] \cdot L[\pi(i)] \right).$$

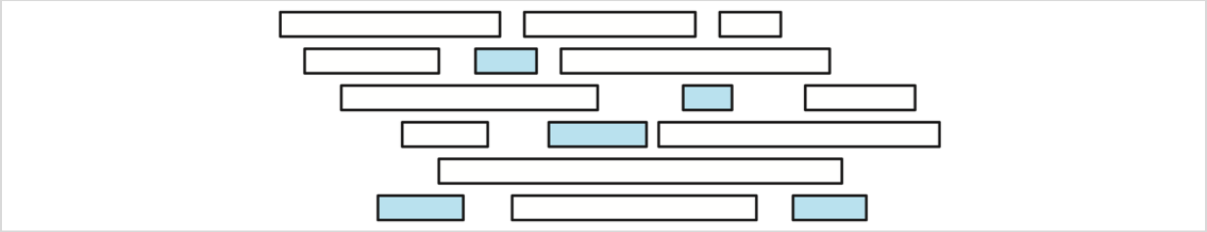
- We should still try to pick a first file and then recursively place the rest, but now what should our first file be?
- Before, we wanted the shortest file first. If all files were the same size, we'd want the most frequently accessed file first so we wouldn't have to move past the other files to access it.
- So let's use both ideas and start with the files minimizing the *ratio* of length divided by frequency.
- Lemma: Let x be a file minimizing L[x] / F[x]. Some optimal placement of files has pi(1) = x.
- Proof:
  - Again, suppose some optimal solution doesn't start with x = pi(i).
  - We'll move x to the first position…
  - Now, any file pi(k) for 1 ≤ k < i just had its total cost for all accesses increase by F[k] * L[x].
  - But the total cost of all accesses to x has now decreased by F[x] * L[k] for all 1 ≤ k < i.
  - In other words the expected cost of pi just changed by
    - sum_{k = 1}^{i - 1} F[k] * L[x] - sum_{k = 1}^{i - 1} F[x] * L[k]
    - = sum_{k = 1}^{i - 1} (F[k] * L[x] - F[x] * L[k]
  - But L[k] / F[k] ≥ L[x] / F[x] implies F[x] * L[k] ≥ F[k] * L[x}, so the cost change is
    - ≤ sum_{k = 1}^{i - 1} 0
    - = 0
  - Again, we can just sort all the files in increasing order of ratio to compute a best permutation in O(n log n) time.

Class Scheduling

- Let's look at a different example. Suppose you're picking classes for next semester, and your number one goal is to graduate as quickly as possible, so you want to take as many courses as possible. Don't worry, these classes you're considering don't require any actual work; you just need to be present for the lecture. By some fluke all classes next semester are scheduled on Mondays only, but you're not allowed to take classes scheduled for conflicting times.
- Formally, let S[1 .. n] be the start time of all n courses offered and F[1 .. n] be their end times; so 0 ≤ S[i] < F[i] for all i.
- You want to find a *maximal conflict-free schedule* which is a maximum size subset X of {1, 2,

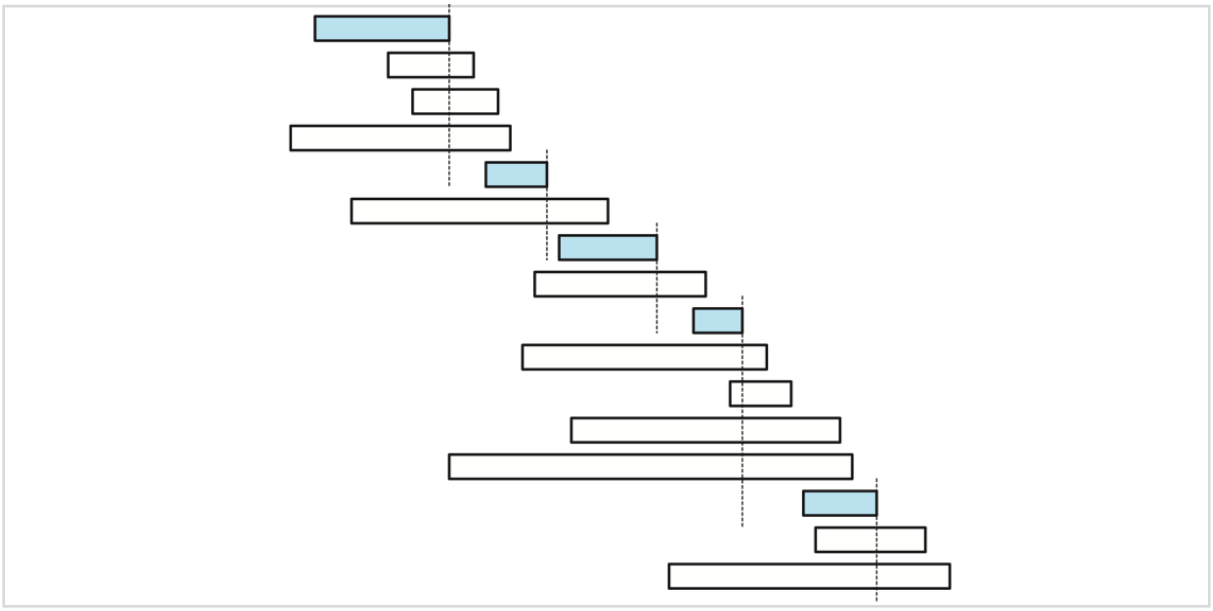
…, n} such that for each i,j in X either S[i] > F[j] or S[j] > F[i].

- Another way to think about it is you have this set of overlapping intervals representing the time span for each course. Find the largest subset of intervals that don't overlap.



- If we were designing a backtracking or dynamic programming algorithm, might loop over all the intervals, trying to guess one to include in our schedule. For each, guess, we'd recursively build the best schedule for the subset of intervals that don't conflict.
- These subsets of classes do have some nice structure to them, so we don't need to consider *every* subset as the input to a recursive subproblem. But even still, we'd at best get an O(n^3) time dynamic programming algorithm using this strategy.
- I'm not going to say more, though, because there's a greedy choice we can make that leads to a much better run time.
- We'll greedily choose the class that finishes first and then recursively build a best schedule for the remaining classes.
- Let's prove this greedy strategy works, and then we'll look at an efficient implementation of the algorithm.
- Lemma: At least one maximal conflict-free schedule includes the class that finishes first.
  - Let f finish first, and consider any maximal conflict-free schedule X.
  - If X includes f, we're done.
  - Otherwise, let g be the first class to finish in X.
  - f finishes before g, so f does not conflict with any classes in X \ {g}.
  - So remove g and replace it with f. The new schedule is just as large and it agrees with our first choice.
- And to be complete, let's argue that our overarching recursive strategy is correct.
  - We know we can start with the class f that finishes first.
  - Given that choice, we cannot take classes conflicting with f.
  - But we can use any subset of non-conflicting classes that don't conflict with f. And by induction, the Recursion Fairy will find the maximal set of such classes.
- We can write our strategy as an iterative algorithm by scanning through the class list ordered by finishing time, and every time we see a new class that doesn't conflict with the last one we chose, taking it. This procedure returns the indices of the classes sorted by finishing time.

```
GreedySchedule(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```



- This figure shows the sorted set of intervals and what we're skipping past.
- For running time, we have the time it takes to sort which is O(n log n). The rest of the algorithm is a simple O(n) time for loop, so the overall running time is O(n log n).

Greedy Algorithms and Exchange Arguments

- We just saw some examples of greedy algorithms.
- Again, not everybody describes it this way, but I like to think of us designing a backtracking algorithm but we pick one "obviously" best choice before recursing instead of enumerating over all possible choices for our first decision.
- Proving our first choice correct requires an exchange argument.
  1. Start with some optimal solution. If it agrees with our first choice, then great! Otherwise…
  2. Do some kind of "exchange" in the optimal solution so it does use our first choice.
  3. Argue that the exchange didn't increase the cost of the solution, so our new solution must still be optimal.
- Sometimes, the new solution ends up being even better than the original one, implying that our first choice *had* to be correct. We'll see an example of that when we do minimum spanning tree algorithms sometime next month.
- Of course, this strategy is still based on backtracking, so you should argue that your recursive call is useful as well.

- And like a lot of things, this is not the *only* way to think about greedy algorithms. Erickson suggests that exchange arguments should try to fix the "first" difference between greedy and optimal solutions, and he offers alternative proofs for the tape storage and class scheduling problems to reflect this viewpoint.
- But I like to teach the backtracking way of thinking about it, because it keeps you all thinking recursively and it solidifies the goal with the exchange argument.
- All that said, you almost never want to use a greedy algorithm if your goal is a provably correct algorithm for a problem. It's extremely rare that the problem is structured so nicely that you can do an exchange argument, and you're better off using dynamic programming. I've had some freedom this semester in what examples I choose for techniques like divide-and-conquer and dynamic programming. Correct greedy algorithms are so rare, that I've already used one of the two examples that appear in pretty much every undergraduate lecture on the topic. Wednesday, I'll describe the other example.