CS 4349 Lecture—September 25th, 2017

Main topics for #lecture include #greedy_algorithms, #tape_storage, and #class_scheduling.

Prelude

- Homework 4 due Wednesday, September 27th.
- Homework 5 will be assigned then, but it won't be due for two weeks, because…
- The Midterm Exam is Wednesday, October 4th from 7:00pm to 9:00pm. **If you know about a legitimate conflict, let me know ASAP, by this Wednesday at the latest.**

Storing Files on Tape

- Suppose we're asked to store n files and lock them in a time capsule for future generations. They're very large, and we want to make sure they're still readable in the near-future, so we're going to use this awesome technology called tape, and even pack the tape reader in the time capsule as well!
- 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, and let pi(i) denote the index of the ith file on tape. Reading file pi(k) would take \sum_{i = 1}^k L[pi(i)] time.
- And the tape reader isn't too advanced, so it resets the head after reading each file. So to read all the files from 1 to n, it will cost sum_{k=1}^n sum_{i=1}^k L[pi(i)].
- So, what order should we store these files so somebody can read all them exactly once as quickly as possible?
- We should store them in increasing order of size!
- Proof:
    - Let pi(i) be the index of the ith file in the best order.
    - Suppose the claim is false so L[pi(i)] > L[p(i+1)] for some i.
    - Let a = pi(i) and b = p(i+1).
    - We could swap files a and b. It would not affect the time to access files before or after a and b.
    - It now costs L[b] more to access to a and L[a] less to access b. So the change in total cost is L[b] - L[a] < 0. We didn't have the best order after all!
- So this gives us an O(n log n) time *greedy algorithm*. Sort the files by size, put the smallest file first, and blindly plow ahead.

- Let's make things a bit more interesting before moving on. Let's say some files are more interesting, funnier, or heartwarming than others. Let F[1 .. n] denote the number of times each file will be accessed.

- So now the total time to do all the reads is sum_{k=1}^n F[pi(k)] sum_{i=1}^n L[pi(i)] = sum_{k=1}^n sum_{i=1}^n (F[pi(k)] * L[pi(i)]).
- Now what order should we store the files?
- One way to think about it is this: We wanted to store the files in increasing order of size. If all the files were the same size, we'd want to store them in *decreasing* order of frequency so the most frequently access files were near where the tape head starts.
- So let's do both and sort the files in increasing order of length / frequency.
- Proof:
    - Let pi(i) be the index of the ith file in the best order.
    - Suppose the claim is false so L[pi(i)] / F[pi(i)] > L[p(i+1)] / F[pi(i+1)] for some i.
    - Let a = pi(i) and b = p(i+1).
    - We could swap files a and b. It would not affect the total time to access files before or after a and b.
    - It now costs F[a] * L[b] more to access a and F[b] * L[a] less to do accesses of b. So the change in total cost is F[a] L[b] - F[b] L[a]. But L[a] / F[a] > L[b] / F[b] => F[b] L[a] > F[a] L[b]. We didn't have the best order after all!
- These proofs used an *exchange argument*. Take a hypothetical optimal solution, find somewhere it differs from the solution we'd like to use, and then do an exchange that makes the solution more like ours but still as good as before the exchange.
- Most proofs of correctness for greedy algorithm use an exchange argument.

Class Scheduling

- Next, let's 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. Yes, it will be a lot of work, but that's not your concern right now. 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.
- You want to find a *maximal conflict-free schedule* which is a largest 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.
- Now, if it was last week, I'd say something about how we need to consider all the first classes we could take and recursively figure out how many classes we can take with our remaining time for each of them. We'd need to write a recurrence, and describe a data

structure, and at the end of it all we'd have an O(n^2) time algorithm.

- But I'm feeling lazier this week. Let's just commit to one first class and deal with the consequences.
- Let's greedily take the class that finishes first and repeat. In other words, we'll scan 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, we'll take it.
- 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]
  ```

- We need to sort by finishing time, which we can do in O(n log n) time using a merge sort. We also have that O(n) time for loop, so the whole thing takes O(n log n) time.
- But does it work? We can again use an exchange argument.
- Claim: There is a maximal conflict-free schedule that includes the class that finishes first.
  - Let f finish first, and let X be a maximal conflict-free schedule. Let g be the first class to finish in X.
  - f does not conflict with any classes in X \ {g} since f finishes even earlier.
  - So remove g and replace it with f. The new schedule is just as large.
- Claim: The greedy schedule is optimal.
  - We can prove it by induction. If there were no classes, then the empty schedule would be optimal.
  - So let's assume the greedy schedule is optimal for n' < n classes.
  - Well, there is an optimal schedule that uses the same first choice as greedy. Given that choice, we cannot take conflicting classes, so our goal is to find a maximal conflict-free scheduled form the non-conflicting classes. Which the greedy algorithm does by our inductive assumption.
- We can do an exchange argument that is a bit more direct.
  - Let <g_1, g_2, .., g_k> be the greedy schedule ordered by class finish times.
  - Let <g_1, g_2, …, g_{i-1}, c_i, c_{i+1}, …, c_m> be an optimal schedule ordered by finish times where c_i is the first class to differ from the greedy schedule and i is as large as possible. If there is no c_i, then the greedy schedule is optimal.
  - Otherwise, g_i finishes before c_i, so it doesn't conflict with c_{i+1} through c_m. So we could have looked at the optimal schedule <g_1, g_2, …, g_{i-1}, g_i, c_{i+1}, …, c_m>, contradicting i being as large as possible.

- Again, we assumed there was an optimal solution that differed from the greedy one, but then we did an exchange to make the optimal solution look more like the greedy one without sacrificing optimality.

Greedy Algorithms Never Work*

- So we saw some examples of greedy algorithms, but here's the thing…
- GREEDY ALGORITHMS NEVER EVER EVER EVER WORK (except when they do)
- Whenever you're in the situation where you want to solve some kind of optimization problem, you'll be tempted to just write a greedy algorithm. But if you really want the best solution, you're probably going to be wrong.
- Very rarely, like with the problems today, there's some easy first choice you can make without any bad consequences. But you'll probably need an exchange argument to know for sure.
- If you don't know for sure, use dynamic programming.