Welcome and Administrivia

Welcome to CSE 541!

• This is a course about algorithms. What does that mean?
• Not a course designed to passively teach specific algorithms. (That was 241/247.)
• Rather, a course for you to learn how to design and analyze your own algorithms.
• Course is designed to teach these skills.
• “Active learning” – no good unless you engage with the material!!!

How is this course structured?

• Class Meetings: explanations and illustrative examples of algorithmic techniques that work for many problems.
• Practice Problems: ungraded problems with solutions to reinforce understanding of techniques
• Homeworks (40%): graded problems without solutions to see if you can apply techniques to new problems
• Exams (60%): final opportunity to demonstrate that you’ve mastered the techniques

What do you need to know right now?

• The course web site is your friend. All assignments and practice problems will appear there.
• Course discussions and communication with instructor and TAs is conducted through our Piazza board.
• Homeworks are submitted electronically via Blackboard. See the Electronic Homework Submission Guide on the website for detailed instructions and help with formatting.
• Read the course overview! It has all the grody administrative details.

• There is a **collaboration policy** for homeworks (not practice probs) – tries to balance fun of collective problem solving with promotion of individual skill building and assessment.
  
  – Please limit group problem solving to at most 3 people at once
  – Nothing written brought to discussion or carried away
  – “Iron Chef Rule” – one hour between discussion and writeups
  – Report (in the comment box of the homework submission interface) any assistance you receive.

Who is here to help you, and how do you contact us?

• TAs: Ray Su, Diqiu Zhou

• To contact myself or TAs, use Piazza.

• Our office hours are all posted on Piazza.

What do I expect you to know already?

• First: how to write a basic proof of correctness and/or running time

• (But I also try to teach this at a more advanced level!)

• Given a problem statement, I *always* expect you to produce
  
  1. An effective method for solving the problem (“algorithm”)
  2. A proof that the algorithm correctly solves the problem
  3. A proof that the algorithm is efficient (running-time analysis)

• Second: asymptotic complexity analysis

• Third: basic data structures: hashing, sorting and searching, binary trees (including results, but not implementation, for balanced trees), elementary graph algorithms

• To warm up and check your grasp of the above skills, there is an ungraded “Homework 0” on the web site. You should be able to do all the problems found there and produce suitable formal proofs.

• You can also use Homework 0 to make sure you know how to submit homework problems through Blackboard.

• If you have doubts about Homework 0 or want me to look at it, please submit it no later than September 8th and let me know (via Piazza) that we should check your solutions.

OK, on to the fun stuff...
2 What Are We Doing Here?

We will focus on *combinatorial optimization* problems

- We need to perform constrained optimization.

- A problem typically includes a set of *constraints* (properties that a valid solution must satisfy) and an *objective function*, or measure of goodness/badness, for solutions.

- Constraints are typically discrete rather than continuous (no such thing as an “almost correct” answer)

- Goal is to find a solution that is
  - *feasible* – satisfies all constraints
  - *optimal* – maximizes (or minimizes) objective

- We want an algorithm that finds an optimal feasible solution for any problem instance.

- Algorithms should be *efficient* (at worst polynomial-time), and faster is better (shoot for at worst quadratic or cubic; linear or \( n \log n \) is strongly preferred if possible)

How better to start than with an example?

3 A Scheduling Problem

- You manage a ginormous space telescope.

- Lots of astronomers want to use it to make observations.

- Each astronomer’s project \( p_i \) requires use of the telescope starting at a fixed time \( s_i \) (when their grant starts) and running for \( \ell_i \) days.

- Only one project can use the telescope at a time.

- Your goal: justify your budget to NASA by scheduling as many projects as possible!

- More formally: given a set \( P \) of projects \( p_i \), each occupying half-open time interval \([s_i, s_i + \ell_i)\),

- Choose a subset \( \Pi \subseteq P \) of projects for which
  - No two projects’ intervals overlap (“conflict”);
  - The number of projects in \( \Pi \) is maximized.

- This is one of many variants of the *scheduling* or *activity selection* problem.

**Example (see attached)**

How should we solve this problem?
• **Suggestion 1**: repeatedly pick shortest non-conflicting, unscheduled project (i.e. that does not conflict with any scheduled project).

• Does this strategy always yield an optimal solution? Prove or disprove.

• **Counterexample:**

• **Suggestion 2**: repeatedly pick non-conflicting project with earliest starting time.

• Does this always yield an optimal solution? Prove or disprove.

• **Counterexample:**

• **Suggestion 3**: first, label each project with number of other projects with which it conflicts. Then, repeatedly pick nonconflicting project with fewest total conflicts.

• Does this always yield an optimal solution? Prove or disprove.

• **Counterexample:**

Aaaaargh! We need a principle to stop the endless flailing!

### 4 An Approach That Works

What structure do all above solutions have in common?

• Repeatedly pick an element until no more feasible choices remain.

• Among all feasible choices, we always pick the one that minimizes or maximizes some property (project length, start time, # conflicts)

• Such algorithms are called greedy.
As we’ve seen, greedy algorithms are frequently not optimal.

Ah, but maybe we have been using the wrong property!

Let’s take another wild guess...

For each project $p_i$, define its finishing time $f_i$ to be $s_i + \ell_i$.

Repeatedly pick non-conflicting, unscheduled project with earliest finishing time.

Here’s a reasonably efficient implementation of this strategy in pseudocode.

```
SCHEDULE(P)
    sort $P$ in increasing order $\{p_1 \ldots p_n\}$ of finishing time $f_i$
    $\Pi \leftarrow \{p_1\}$
    $j \leftarrow 1$
    for $i$ in 2..n do
        if $s_i \geq f_j$
            $\Pi \leftarrow \Pi \cup \{p_i\}$
            $j \leftarrow i$
    return $\Pi$
```

Sorting the times requires $O(n \log n)$ time.

Selection procedure takes $O(1)$ time for each $i$, so $O(n)$ overall.

Hence, total complexity of SCHEDULE is $O(n \log n)$.

But does it work????

5 Proving Correctness

Why should this greedy algorithm work when all others failed? Three key observations do it for us:

1. **Greedy Choice**: For every problem instance $P$, there exists an optimal solution that includes first element $\hat{p}$ picked by greedy algo.

2. **Inductive Structure**: After making greedy first choice $\hat{p}$ for problem instance $P$, we are left with smaller subproblem $P'$, such that, if $\Pi'$ is a feasible solution to $P'$, then $\Pi' \cup \{\hat{p}\}$ is a feasible solution to $P$.

   (Colloquially, we say that subproblem $P'$ has no external constraints restricting its feasible solutions.)

3. **Optimal Substructure**: If $P'$ is subproblem left from $P$ after greedy choice $\{\hat{p}\}$, and $\Pi'$ is an optimal solution to $P'$, then $\Pi' \cup \{\hat{p}\}$ is an optimal solution to $P$.

Let’s prove these properties for SCHEDULE’s greedy choice.

- **Greedy Choice**: Let $P$ be instance of scheduling problem, and let $\hat{p} \in P$ be first project picked by SCHEDULE. Then there exists an optimal solution to $P$ that contains $\hat{p}$.
• Pf: we use an exchange argument.
• Let $\Pi^*$ be any optimal solution to $P$.
• If $\hat{p} \in \Pi^*$, we are done.
• Otherwise, let $\Pi'$ be solution obtained by removing earliest project $p \in \Pi^*$ and adding $\hat{p}$.
• By construction, $\hat{p}$ ends no later than $p$, so if $p$ does not conflict with any later project of $\Pi^*$, neither does $\hat{p}$. Hence, $\Pi'$ is feasible.

Moreover, $|\Pi'| = |\Pi^*|$, so $\Pi^*$ is optimal. QED

One down, two to go.

• Inductive Structure: After making greedy first choice $\hat{p}$ for problem instance $P$, we are left with smaller subproblem $P'$, with no external constraints.
• After we select the first project $\hat{p}$, what is remaining subproblem?
• It’s not just $P - \{\hat{p}\}$!
• Having selected $\hat{p}$, we cannot pick any other project that conflicts with it.
• Put another way, choosing $\hat{p}$ imposes an external constraint on subproblem $P - \{\hat{p}\}$, because not every feasible solution to the subproblem can be combined with the greedy choice.
• Simple fix: define the subproblem to be

\[ P' = P - \{\hat{p}\} - \{\text{projects that conflict with } \hat{p}\}. \]

Now any feasible solution to $P'$ can feasibly be combined with $\hat{p}$, and so there is no external constraint on $P'$.

Two down, one to go.

• Optimal Substructure: If $\Pi'$ is an optimal solution to subproblem $P'$, then $\Pi' \cup \{\hat{p}\}$ is an optimal solution to $P$.
• Pf: let $\Pi'$ be as given.
• Then $\Pi = \Pi' \cup \{\hat{p}\}$ is a feasible solution to $P$, with size $|\Pi| = |\Pi'| + 1$.
• Now suppose $\Pi$ were not optimal.
• Let $\Pi'^*$ be an optimal solution containing $\hat{p}$. (Such a solution must exist by the Greedy Choice Property.)
Then \( \Pi^* - \{\hat{p}\} \) is a feasible schedule for \( P' \) with
\[
|\Pi^* - \{\hat{p}\}| > |\Pi - \{\hat{p}\}| = |\Pi'|,
\]
contradicting optimality of \( \Pi' \).

- Conclude that \( \Pi \) must be optimal. QED

OK, that was fun. But why do these three facts constitute a proof that \texttt{Schedule} always obtains an optimal solution?

- **Claim**: \texttt{Schedule}'s solution is optimal for every problem instance \( P \).
- **Pf**: by induction on size of problem \( P \).
- **Bas**: if \( P \) has size 1, greedy solution is trivially as good as optimal (it picks the one element).
- **Ind**: suppose \texttt{Schedule}'s solution is optimal for problem instances of size \(< k \).
  - Consider an instance \( P \) of size \( k \).
  - Let \( P' \) be subproblem obtained from \( P \) after making first greedy choice, and let \( \hat{p} \) be this choice. Observe that \( |P'| < |P| \).
  - By IH, \texttt{Schedule} produces an optimal feasible solution \( \Pi' \) for \( P' \). (This claim holds by inductive structure property.)
  - Inductive structure and optimal substructure properties guarantees that \( \Pi' \cup \{\hat{p}\} \) is an optimal feasible solution for \( P \).
  - Hence, \texttt{Schedule} optimally solves \( P \) of size \( k \). QED

**Key Observation**: the inductive proof invokes the two structural properties as “subroutines”. The optimal substructure property in turn uses the greedy choice property in its proof. This form of argument is a “design pattern” for proving correctness of a greedy algorithm. It also serves as a guide to algorithm design: pick your greedy choice to satisfy G.C.P. while leaving behind a subproblem with optimal substructure!