===
CSC 373   Lecture Summary for Week 1   Winter 2019
===

Introduction

Course Information Sheet: waiting for last details to be ironed out, will be distributed later this week; go over main points (marking scheme, textbook).

Problem: collect raw petrol from 100 oil platforms.
(For runtime, assume 10^9 ops/sec.)

a. Use a tanker
   - parameters: cost(A,B) for any two platforms A, B -- could be different from distance (natural obstacles, etc.)
   - best algorithm: over $4*10^{13}$ years! (more than 2^{100} ops / 10^9 ops/sec / 31536000 sec/year = 40196936841331 years)
b. Use pipelines
   - parameters: cost(A,B) for any two platforms A, B; no junctions outside platforms
   - best known algorithm: approx. 10 micro-seconds (100^2 ops)
c. Add constraint: cannot connect more than 4 pipelines at each platform (or any other fixed constant k instead of 4)
   - best known algorithm: approx. $4*10^{13}$ years
d. Ease constraint: allow junctions outside platforms (keep cap on maximum number of pipelines allowed at each junction)
   - best known algorithm: still approx. $4*10^{13}$ years
e. Back to tanker: relax requirements to _approximate_ smallest total cost, i.e., cost within factor K of best
   - best known algorithm: still approx. $4*10^{13}$ years, irrespective of value of K
f. Add constraint: cost satisfies triangle inequality: for any three platforms A, B, C, cost(A,C) <= cost(A,B) + cost(B,C) (e.g., if cost is directly proportional to distance)
   - best known algorithm for relaxed requirement: approx. 1 second

What's the deal?
  - P: class of problems that have polynomial-time (i.e., "efficient") algorithmic solutions
  - NP-hard: class of problems for which no efficient algorithm is known (only known algorithms are exponential time)

Vast majority of real-world problems fall into one of these two classes (P or NP-hard). Important to recognize problems in each class and to handle both kinds of problems appropriately.

In this course:
  - techniques for writing efficient algorithms for problems in P;
  - techniques for deciding whether a problem is in P or NP-hard;
  - techniques for handling NP-hard problems.

Background (from CSC263 and its prerequisites):
  - Asymptotic notation (big-Oh, Ω, Θ), analysis of runtimes for iterative and recursive algorithms.
  - Data structures: queues, stacks, hashing, balanced search trees, priority queues, heaps, union-find/disjoint sets.
  - Graphs: definitions, properties, traversal algos (BFS, DFS).
  - Induction and other proof techniques, proving correctness of iterative and recursive algorithms.

```
-----------------
Greedy Algorithms
-----------------

"At each step, make the choice that seems best at the time; never change
your mind."

Activity Scheduling.
    Input: Activities A_1, A_2, ..., A_n. Each activity A_i consists of
        positive integer start time s_i and finish time f_i (s_i < f_i).
    Output: Subset of activities S such that all activities are "compatible"
        (no two of them overlap in time) and |S| is maximum.

TERMINOLOGY:
    In general, for maximization problem with solution S worth val(S),
  - "maximAL" = nothing can be added to S to increase val(S);
  - "maximUM" = no solution has larger value.

 A. Brute force: consider each subset of activities.
    Correctness? Trivial.
    Runtime? \Omega(2^n), not practical.

 B. Greedy by start time:
        sort activities s.t. s_1 <= s_2 <= ... <= s_n
        S := {}  # partial schedule
        f := 0  # last finish time of activities in S
        for i in [1,2,...,n]:
            if f <= s_i:  # A_i is compatible with S
                S := S U {A_i}
                f := f_i
        return S
    Runtime? Sorting is \Theta(n log n), main loop is \Theta(n).
        Total is \Theta(n log n).
    Correctness? Doesn't work. Counter-example:
        |------------------------------|
          |---| |---| |---| ... |---|

 C. Greedy by duration:
        similar to above except sort by nondecreasing duration, i.e.,
        f_1-s_1 <= f_2-s_2 <= ... <= f_n-s_n
    Correctness? Counter-example:
        |-----| |-----|  |-----| |-----| ... |-----| |-----|
             |---|            |---|               |---|

 D. Greedy by overlap count:
        similar to above except sort from fewest conflicts to most conflicts
        ("conflict" = overlap with some other activity)
    Correctness? Counter-example:
        |---| |---| |---| |---| ... |---| |---| |---| |---|
           |---| |---| |---|           |---| |---| |---|
           |---|       |---|           |---|       |---|
           |---|       |---|           |---|       |---|

 E. Greedy by finish time:
        similar to above except sort by nondecreasing finish time, i.e.,
        f_1 <= f_2 <= ... <= f_n
    Correctness? No counter-example...

  - Intuition: algorithm picks activities that "free up" resources as early
```

as posible. BUT: intuition for others also made sense...

- How to tell if this works? Will show general technique for proving correctness of greedy algorithms.

- Let S_0, S_1, ..., S_n = partial solutions constructed by algo. at the end of each iteration.

- Two possibilities:
    . Prove each S_i is optimal solution to sub-problem.
      Works for some problems, but does not generalize well (some problems don't decompose into sub-problems naturally).
    . Prove each S_i can be "completed" to reach optimal solution.
      Can be trickier but generalizes well.

- Say S_i is "promising" if there is some optimal solution OPT that *extends* S_i using only activities from {A_{i+1},...,A_n} (i.e., S_i (_ OPT (_ S_i U {A_{i+1},...,A_n}).
  Note: OPT may not be unique (there may be more than one way to achieve optimal).

- Prove that "S_i is promising" is a loop invariant, by induction in i (number of iterations).

    . Base case: S_0 = {}: any optimal solution OPT extends S_0 using only activities from {A_1,...,A_n}.

    . Ind. Hyp.: Suppose i >= 0 and optimal OPT extends S_i using only activities from {A_{i+1},...,A)n}.

    . Ind. Step: To prove: S_{i+1} is promising w.r.t. {A_{i+2},...,A_n}.
      From S_i to S_{i+1}, algo. either rejects or includes A_{i+1}.

      Case 1: S_{i+1} = S_i
          This means A_{i+1} not compatible with S_i. Since OPT includes S_i, A_{i+1} is also incompatible with OPT.
          Then OPT extends S_{i+1} using only activities from {A_{i+2},...,A_n} (since S_i (_ OPT and A_{i+1} not compatible with S_i (_ OPT).

      Case 2: S_{i+1} = S_i U {A_{i+1}}
          OPT may or may not include A_{i+1}, so consider both possibilities.

          Subcase 2.1: A_{i+1} (- OPT
              Then OPT already extends S_{i+1} using only activities from {A_{i+2},...,A_n}.

  NOTE: Every case and subcase so far holds no matter how the activities are sorted initially -- in other words, our proof does not yet depend on the ordering. But we know this is important: it comes into the next subcase.

          Subcase 2.2: A_{i+1} !(- OPT
              How can this happen? There must be A_j (- OPT that overlaps with A_{i+1} (otherwise, OPT U A_{i+1} would be better than optimal OPT). Also, j > i+1 because A_{i+1} is compatible with S_i, so f_j >= f_{i+1} and at most one A_j overlaps A_{i+1} (otherwise OPT would contain overlapping activities, or an activity outside S_i with finish time earlier than

A_{i+1}, both impossible).
But then, OPT' = OPT U {A_{i+1}} - {A_j} extends S_{i+1} using {A_{i+2},...,A_n}: same number of activities as OPT, and no overlap introduced because f_{i+1} <= f_j.

NOTE: Argument above known as "exchange lemma": arguing that any optimal solution can be made to agree with greedy solution, one element at a time. Just like definition of "extends", every problem and algorithm yields a different "exchange lemma" -- there is no single Exchange Lemma that applies to every algorithm and problem!

In all cases, there is some optimal OPT' that extends S_{i+1} using only activities from {A_{i+2},...,A_n}.

- So each S_i is promising. In particular, S_n is promising, i.e., there is optimal OPT that "extends" S_n using only activities from {}. In other words, S_n is optimal.

For Next Week

* Readings: Sections 5.1, 4.4.
* Self-Test: Exercises 5.1, 5.2, 4.1.