Introduction

Course Info sheet

Background:

- Asymptotic notation ($\mathcal{O}$, Ω, Θ) [Chapter 0 in the textbook], analysis of runtimes for iterative and recursive algorithms [Chapter 2], and the Master Theorem.
- Data structures: queues, stacks, hashing, balanced search trees, priority queues, heaps, union-find/disjoint sets.
- Graphs: definitions, properties, traversal algos (BFS, DFS). [Ch 3, 4.2]
- Induction and other proof techniques, proving correctness of iterative and recursive algorithms.

A very brief review:

Master Theorem (Chapter 2.2): If $T(n) = aT(\lceil n/b \rceil) + \mathcal{O}(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geqslant 0$, then

$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{if } d > \log_b a \\ \mathcal{O}(n^d \log n) & \text{if } d = \log_b a \\ \mathcal{O}(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Worst-case running time: For any algorithm A, worst-case running time of A is $T_A(n)$ = maximum # steps executed by A over all inputs of size n. Definition of *input size* is problem-dependent but must be reasonable, i.e., within a constant factor of actual *bit-size* of input for any reasonable implementation.

Upper bounds: To prove $T(n)$ is $O(f(n))$, give a general argument that algorithm never takes more than $c \times f(n)$ steps for all inputs of size n (for some constant c). Proving this for a single input is not enough, unless you also prove that this input is the worst-case (which requires reasoning about all inputs of size n anyway).

Lower bounds: To prove $T(n)$ is $\Omega(f(n))$, produce a single input of size n for which algorithm takes at least $c \times f(n)$ steps (for some constant c). It's OK to prove this for all inputs, although that won't always work. For example, linear search takes worst-case time $\geqslant n$ (when the element being searched for is not in the list), but it does NOT take time $\geqslant n$ for every input.

Tight bounds: To prove $T(n)$ is $\Theta(f(n))$, prove $T(n)$ is $O(f(n))$ and $T(n)$ is $\Omega(f(n))$, as above.

DFS (depth-first search): Traverse the graph from one node (call it the root) and fully explore each branch before backtracking.

BFS (breath-first search): Traverse the graph from one node (call it the root) and explore all neighboring nodes. For each neighboring node (one after the other) explore its neighboring nodes. Continue until all nodes are visited.

Example: Suppose 10^9 ops/sec. Collect raw petrol from 100 oil platforms.

a. Use a tanker

- parameters: $cost(A, B)$ for any two platforms A, B
- could be different from distance (natural obstacles, etc.)
- best known algorithm: approx. 2^{100} operations. approx. 4×10^{13} years.

b. Use pipelines

- parameters: $cost(A, B)$ for any two platforms A, B; no junctions outside platforms
- best known algorithm: approx. 100^2 operations. approx 10 micro-seconds.

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

NOTE: There are many problems with efficient algorithms such that a change may make them very hard (exponential time complexity). For example:

- Interval Scheduling vs Job Interval Scheduling
- Minimum Spanning Tree (MST) vs Bounded degree MST
- MST vs Steiner tree
- Shortest paths vs Longest paths
- 2-Colourability vs 3-Colourability

Problem Classes

- 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)
- Important to recognize problems in each class and to handle both kinds of problems appropriately.

In this course we learn:

- 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.

GREEDY ALGORITHMS:

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

Example (Interval Scheduling Problem)

Input: Intervals $I_1, I_2, ..., I_n$. Each interval I_i consists of positive integer start time s_i and finish time f_i ($s_i < f_i$).

Output: Subset of intervals S such that all intervals are *compatible* (no two of them overlap in time) and $|S|$ is maximum.

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

B. Greedy by start time :

Algorithm 1: Greedy by start time

```
1 Sort intervals such that $s_1 \leqslant s_2 \leqslant \ldots \leqslant s_n$
2 $S := \{\}$                                  // partial schedule
3 $f := 0$                                     // last finish time of intervals in $S$
4 for $i$ in $[1, 2, ..., n]$ do
5     if $f \leqslant s_i$                      // $I_i$ is compatible with $S$
6     then
7         $S := S \cup \{I_i\}$
8         $f := f_i$
9 return $S$
```

Correctness? Doesn't work. Counter-example:

```
|-------------------------|
  |---| |---| |---| |---|
```

C. Greedy by duration : similar to above except sort by nondecreasing duration, i.e., $f_1 - s_1 \leqslant f_2 - s_2 \leqslant \cdots \leqslant f_n - s_n$ and change line 5 to "if I_i is compatible with all intervals in S"

Correctness? Counter-example:

```
|-----| |-----|
     |---|
```

D. Greedy by overlap count : similar to above except sort from fewest *conflicts* to most *conflicts* (conflict: overlap with some other interval)

Correctness? Counter-example:

```
|---| |---| |---| |---|
   |---| |---| |---|
   |---|       |---|
   |---|       |---|
```

E. Greedy by finish time : similar to above except sort by nondecreasing finish time, i.e.,

$$f_1 \leqslant f_2 \leqslant ... \leqslant f_n$$

Correctness? No counter-example $\cdots$

The exchange proof

- Intuition: algorithm picks intervals that *free up* resources as early as possible. BUT: intuition for others also made sense $\cdots$
- How to tell if this works? Will show general technique for proving correctness of greedy algorithms.
- Let $S_0, S_1, ..., S_n$ be the partial solutions constructed by the algorithm 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 S'_i that *extends* S_i using only intervals from $I_{i+1}, ..., I_n$ (i.e., $S_i \subseteq S'_i \subseteq S_i \cup \{I_{i+1}, ..., I_n\}$). Note: S'_i 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 S'_0 extends S_0 using only intervals from $\{I_1, ..., I_n\}$.
  - Ind. Hyp.: Suppose $i \geqslant 0$ and optimal S'_i extends S_i using only intervals from $\{I_{i+1}, ..., I_n\}$.
  - Ind. Step (To prove): S_{i+1} is promising w.r.t. $\{I_{i+2}, ..., I_n\}$.
    From S_i to S_{i+1}, the algorithm either rejects or includes I_{i+1}.
    * Case 1: $S_{i+1} = S_i$
      This means I_{i+1} is not compatible with S_i. Set $S'_{i+1} = S'_i$. Then S'_{i+1} extends S_{i+1} using only intervals from $\{I_{i+2}, ..., I_n\}$ (since $S_i \subseteq S'_i$ and I_{i+1} is not compatible with S_i).
    * Case 2: $S_{i+1} = S_i \cup \{I_{i+1}\}$
      S'_i may or may not include I_{i+1}, so consider both possibilities.
      · Subcase a: $I_{i+1} \in S'_i$
        Then set $S'_{i+1} = S'_i$, so S'_{i+1} extends S_{i+1} using only intervals from $\{I_{i+2}, ..., I_n\}$.
      · Subcase b: $I_{i+1} \notin S'_i$
        How can this happen? There must be $I_j \in S'_i$ that overlaps with I_{i+1} (otherwise, $S'_i \cup I_{i+1}$ would be better than optimal S'_i).
        $j > i+1$ because S'_i only uses intervals from $\{I_{i+1}, ..., I_n\}$ to extend S_i. Because of sorting order, this means $f_{i+1} \leqslant f_j$.
        Also at most one I_j overlaps I_{i+1} (otherwise, assume there are two intervals that overlap I_{i+1}. Call them I_{j_1} and I_{j_2}. Since $f_{i+1} \leqslant f_{j_1}$ and $f_{i+1} \leqslant f_{j_2}$, intervals I_{j_1} and I_{j_2} overlap. Thus S'_i would contain overlapping intervals that is a contradiction).
        Therefore, $S'^*_i = S'_i \cup \{I_{i+1}\} - \{I_j\}$ also extends S_i using $\{I_{i+1}, ..., I_n\}$ (it contains the same number of intervals as S'_i, and no overlap introduced).
        Set $S'_{i+1} = S'^*_i$. Then $S'{}_{i+1}$ extends S_{i+1} using $\{I_{i+2}, ..., I_n\}$.

        Argument above known as *exchange lemma*: arguing that any optimal solution can be made to agree with greedy solution, one element at a time.

    In all cases, there is some optimal S'_{i+1} that extends S_{i+1} using only intervals from $I_{i+2}, ..., I_n$.
  - So each S_i is promising. In particular, S_n is promising, i.e., there is optimal S'_n that *extends* S_n using only intervals from $\{\}$. In other words, S_n is optimal.

The charging proof (will be explained next week)

- The idea is to charge any interval in an optimal solution to a unique interval in the greedy solution.
- Let OPT be an optimal solution.
- Let S be the greedy solution (the solution identified by the greedy algorithm EFT).
- Define a 1-1 map (a 1-1 function) $h : OPT \rightarrow S$. If such a map exists, we can conclude that $|OPT| \leqslant |S|$.
- Definition of h: $h(J)$ is the interval $J' \in S$ with two properties: (1) it intersects J, and (2) it has the earliest finishing time amongst intervals in S intersecting J.
  - First, h is a function (i.e., J' must exist and is unique). Why?

  - Second, h is 1-1. Assume two intervals $J_1, J_2 \in OPT$ are mapped to the same interval $J \in S$. WLOG, assume that $f_1 < f_2$. Let f be the finishing time of J. By the denition of the mapping h, $f \leqslant f_1$ or else the greedy EFT algorithm would have taken J_1 (and not J). So we have $f \leqslant f_1 < f_2$ and since J_1 and J_2 cannot intersect, J_2 cannot intersect J'.

- Thus, $|OPT| \leqslant |S|$. Therefore, S is optimal.