The "new" 6.046

6.006 prerequisite
    Data structures such as heaps, trees & graphs
    Algorithms for sorting, shortest paths, graph search, dynamic programming

Seven modules
    Divide & Conquer - FFT, randomized algs
    Data structures - amortized d.s.
    Optimization - greedy, dynamic prog.
    Network Flow
    Linear Programming
    Intractability (& dealing with it)
    Advanced topics

Handouts: course information, objectives, PS1
Register on Stellar website for 6.046 for sections on Friday
Pay particular attention to course collaboration policy!
The theme of today's lecture

Very similar problems can have very different complexity.

Recall: P: class of problems solvable in polynomial time. $O(n^k)$ for some constant $k$
Shortest paths in a graph $O(V^2)$ e.g.

NP: class of problems verifiable in polynomial time.

Hamiltonian cycle a directed graph $G(V,E)$ is a simple cycle that contains each vertex in $V$.

Determining whether a graph has a Hamiltonian cycle is NP-complete but verifying that a cycle is Hamiltonian is easy.

$P \subseteq NP$ but is $P = NP$?

NP-complete: problem is in NP and is as hard as any problem in NP.

If any NPC problem can be solved in poly time, then every problem in NP has a poly time solution.
Interval Scheduling

Resources & requests
Requests $1, \ldots, n$, single resource
$S(i)$ start time, $f(i)$ finish time $S(i) < f(i)$

Two requests $i$ & $j$ are compatible if they don't overlap, i.e., $f(i) \leq S(j)$ or $f(j) \leq S(i)$

Goal: select a compatible subset of requests of maximum size.

Claim: We can solve this using a greedy algorithm.
A greedy algorithm is a myopic algorithm that processes the input one piece at a time with no apparent look ahead.
Greedy Interval Scheduling

Use a simple rule to select a request $i$.
Reject all requests incompatible with $i$.
Repeat until all requests are processed.

Possible rules?

1. Select request that starts earliest, i.e., minimum $s(i)$
   \[ \text{long one is earliest.} \]
   \[ \text{bad!} \]

2. Select request that is smallest, i.e., minimum $f(i) - s(i)$
   \[ \text{smallest.} \]
   \[ \text{bad!} \]

3. For each request find # incompatibles.
   Select the one with minimum # incompatibles.
   \[ \text{bad selection!} \]

4. Select request with earliest finish time, i.e., minimum $f(i)$
Exercise: Prove greedy algorithm based on earliest finish time is optimal.

Sketch: Assume optimal solution $O$. Greedy produces $A$. Show $|A| = |O|$, by showing greedy algo "stays ahead".

Lemma: For all indices $r \leq k$, $f(i_r) \leq f(r)$. Prove using induction.

Complexity? Sort in terms of earliest finishing time $O(n \log n)$

Related problem: Interval Partitioning

Schedule all requests using a minimum number of identical resources.

Similar algo works. Sort in terms of earliest finish times. Use existing resources if possible, else use a new/available one.
Weighted Interval Scheduling

Each request $i$ has a weight $w(i)$
Schedule subset of requests with maximum weight.

Dynamic Programming

Sort requests in earliest finish time order
$f(1) \leq f(2) \ldots \leq f(n)$

Define $p(j)$ for interval $j$ to be the largest index $i < j$ such that $i$ & $j$ are compatible.

$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$
**DP Guessing**

Consider optimal solution $O$. Either $n$ belongs to $O$, or it does not.

If $n \notin O$, intervals $p(n)+1, \ldots, n-1$ all overlap with $n$ and cannot belong to $O$.

$O$ must include an optimal solution to problem consisting of requests $\{1, \ldots, p(n)\}$.

If $n \notin O$, then $O$ is simply equal to the problem $\{1, 2, \ldots, n-1\}$ optimal solution.

**DP Solution**

Can recurse & memoize, here’s an iterative version.

Array $M[0..n]$ holds optimal solution’s values.

\[
M[0] = 0
\]

for $j = 1, 2, \ldots, n$

\[
M[j] = \max\left( w(j) + M[p(j)], M[j-1] \right)
\]

Once we have $M$ trace back to find optimal solution in $O(n)$ time. (Sorting initially takes $O(n \log n)$.)

See appendix.
NON-IDENTICAL MACHINES.

requests 1, ..., n, s(i), f(i) as before
m machine types \( \mathcal{T} = \{ T_1, \ldots, T_m \} \)
weight of 1 for each request.

\( Q(i) \subseteq \mathcal{T} \) is set of machines that request i can be serviced on.

Maximize the number of jobs that can be scheduled on the m machines.

NP can clearly check that any given subset of jobs with machine assignments is legal.

Can k ≤ n request be scheduled? NP-complete

Maximum request should be scheduled. NP-hard.

Dealing with Intractability

1) Approximation algorithms: Guarantee within some factor of optimal in poly time.
2) Pruning heuristics to reduce (possibly exp) runtime on "real-world" examples
3) Greedy or other suboptimal heuristics that work well in practice → no guarantees
Reductions

For each request $i$ and each machine $j$, Boolean variable $x_{ij}$ indicates whether request $i$ is scheduled on machine $j$. If request $i$ cannot be scheduled on machine $j$, i.e. $j \notin Q(i)$, set $x_{ij} = 0$.

$S = \text{set of start times} = \{s(i)\}$

For $t \in S$, $R(t)$ is set of requests containing $t$.

Each job scheduled at most once.

Each machine executes at most one job at each time point.

Integer linear programming

$\text{Max} \sum_{i=1}^{n} \sum_{j \in Q(i)} w(i) x_{ij}$

$s.t.$

$\forall i \sum_{j=1}^{n} x_{ij} \leq 1$

$\forall j \sum_{i \in R(t)} x_{ij} \leq 1$

$\forall i, j \ x_{ij} \in \{0, 1\}$

$0 \leq x_{ij} \leq 1$

Linear programming hard!

$= 1$ in unweighted case

Poly time!
Strategies

2) Run ILP solver (e.g. CPLEX) that incorporates pruning heuristics to reduce runtime & provides optimal solution.
   - might not finish.

1) LP can be solved in poly time.
   But $x_{ij}$ may be fractional in solution
   
   Round up or down LP solution and show that solution is not too far off from optimal $\leq$ approximation algo.

   Bhatia et al. (1 - 1/e) approximation algo.
Find-Solution \( j \)

if \( j = 0 \) then
  Output nothing

else
  if \( w(j) + M[p(j)] > M[j-1] \) then
    Output \( j \) together with Find-Solution\( (p(j)) \)
  else
    Output Find-Solution \( (j-1) \)