CSE 417
Course Review
Reminders

> HW9 due today

> Please fill out course evaluations

> Final on Monday, 2:30–4:20pm
  – will assume familiarity with HW assignments
    > (otherwise, no memorization... I'll remind if necessary)
  – be prepared to apply all techniques to new problems
Course Goal

- Teach you techniques that you can use to create new algorithms **in practice** when the opportunity arises
  - (or in coding interviews)
  - they will also help you understand existing algorithms
Course Topics

Design Techniques
1. Divide & Conquer
2. Dynamic Programming
3. Branch & Bound

Modeling Techniques
1. Shortest Paths
2. Binary Search
3. Network Flows
Course Topics

Design Techniques
1. Divide & Conquer
2. Dynamic Programming
3. Branch & Bound

Techniques that you can apply to design new algorithms
– each of these has a good chance of being useful in practice
Course Topics

Modeling Techniques
1. Shortest Paths
2. Binary Search
3. Network Flows

Solve new problems by transforming them into familiar ones
– these three are the most likely to show up in practice
– learning to recognize them is a useful skill
Course Topics

Design Techniques
1. Divide & Conquer
2. Dynamic Programming
3. Branch & Bound

Modeling Techniques
1. Shortest Paths
2. Binary Search
3. Network Flows

Q: How do I know which technique to use?
A: You don’t need to. Just try them all
   – in practice, you have plenty of time to do this
   – (interviews & tests have artificially restricted time)
Outline

- Binary Search
- Divide & Conquer
- Dynamic Programming
- Network Flows
- Branch & Bound
Binary Search

> Search a space of size $U$ in $O(\log U)$ time by repeatedly removing a constant fraction of the space

> If $U$ is polynomial (e.g., $n^5$), then search is inconsequential
> If $U$ is exponential (e.g., $2^n$), then search is polynomial time

> Applications:
  – find element in a sorted list
    > more generally, invert a monotonic function
  – find the min / max of a unimodal function
Tool #1: Binary Search

> **Problem**: given inputs $(\text{input}_1, ..., \text{input}_k)$, compute some output
  >  i.e., compute a function $(\text{input}_1, ..., \text{input}_k) \rightarrow \text{output}$

> **Ask**: would it be easier to compute $(\text{input}_1, ..., \text{output}) \rightarrow \text{input}_k$?

> **Ask**: is that function monotonic?

> If so, then we can solve problem with binary search
  >  define $f : (\text{input}_1, ..., \text{input}_{k-1}, \text{output}) \rightarrow \text{input}_k$
  >  binary search over output parameter to find where $f$ equals $\text{input}_k$
Example: given costs $A, B_S, M_S, H$, and revenue, compute the hemming cost $H$ such that min manufacturing cost $= \text{revenue}$

Q: Would it be easier to compute $(A, B_S, M_S, H) \rightarrow \text{min cost}$?
Q: Is this function monotonic in $H$?

Yes (both). So use binary search
Outline

> Binary Search
> Divide & Conquer
> Dynamic Programming
> Network Flows
> Branch & Bound
Divide & Conquer

Algorithmic approach:

1. **Divide** the input data into 2+ parts
2. **Recursively** solve the problem on each part
   - i.e., solve the *same problem* on each part
3. **Combine** those solutions to solve the original problem
Tool #2: Divide & Conquer

> **Ask**: would having the solutions to sub-problems on two halves of the data allow me to solve the problem?

> When finished, consider whether divide + combine is truly an easier problem
>  – may now realize a faster way to solve it directly
**Divide & Conquer**

- Example: merge sort
  - can easily merge in $O(n)$ time
  - whereas obvious algorithms for sorting take $O(n^2)$ time
  - divide & conquer gives an $O(n \log n)$ algorithm

- Example: counting inversions (i.e., indexes $i < j$ with $A[i] > A[j]$)
  - find inversions in $A[1 .. n/2]$ and $A[n/2+1 .. n]$
  - just need to find inversions with $i \leq n/2$ and $n/2 < j$
    - for each on the left, count those on the right that are smaller
    - easy if you sort the right half first... can then binary search
    - ... or use a two-finger algorithm (which is actually merge sort)
Divide & Conquer

> Applications:
  – **sorting**: merge sort, quick sort (& quick select)
  – **multiplication**: integers, matrices, FFT
  – **geometry**: Voronoi diagrams, closest pair of points
Master theorem gives the running time for almost any example
  - compare number of leaves in recursion tree to time for split + combine
  - if one asymptotically dominates the other, that is the running time
  - otherwise, running time is that times $O(\log n)$
Outline

- Binary Search
- Divide & Conquer
- Dynamic Programming
- Network Flows
- Branch & Bound
**Dynamic Programming**

Algorithmic approach:

1. Solve problem using solutions to *any* sub-problems
   - (generalization of Divide & Conquer)

2. Determine all sub-problems necessary to apply this recursively

3. Count the total number of such sub-problems
   - needs to be polynomial
Tool #3: Dynamic Programming

> **Ask**: how could the optimal solution use the last element of input?
  - for each possibility, describe the rest of the optimal solution (without the last input) as the optimal solution of a sub-problem
    > this is the optimal sub-structure...
    > the fact that the optimal overall solution is also optimal on at least one particular sub-problem is the reason we can find it efficiently

> **A common case**: solutions are subsets
  - optimal subset could include the last element or not
  - if not, must be optimal subset of items 1 .. n-1
  - if so, must be optimal subset of items 1 ... n-1 to which item n can be *legally added*
Dynamic Programming

> Example: Knapsack
  - optimal solution either includes last item \((w_n, v_n)\) or it does not
  - if not, it is also optimal on \((w_1, v_1), (w_{n-1}, v_{n-1})\) with weight limit \(W\)
  - if so, it is also optimal on \((w_1, v_1), (w_{n-1}, v_{n-1})\) with weight limit \(W - w_n\)

> Example: Longest Common Subsequence
  - optimal solution might use just \(a_n\), just \(b_m\), both or neither
  - if no \(a_n\), rest is optimal on \(a_1, \ldots, a_{n-1}\) and \(b_1, \ldots, b_m\)
  - if no \(b_m\), rest is optimal on \(a_1, \ldots, a_n\) and \(b_1, \ldots, b_{m-1}\)
  - if both, rest is optimal on \(a_1, \ldots, a_{n-1}\) and \(b_1, \ldots, b_{m-1}\)
    > length of opt is 1 + length from sub-problem
Dynamic Programming

Applications:

- **ML**: speech recognition, parsing natural language
- **graphics**: optimal polygon triangulation
- **compilers**: parsing, optimal code generation
- **databases**: query optimization
- **networking**: routing
- Practical applications:
  - spell checking
  - file comparison
  - document layout
  - pattern matching
Dynamic Programming

> Extremely useful for finding optimal trees
  – optimal BST
  – matrix chain multiplication (secretly a parse tree)
  – optimal polygon triangulation (secretly a tree with edges as leaves)
  – CKY parsing

> Example of asking what the solution looks like in general rather than how it uses the last input
Dynamic Programming

- Finding the optimal substructure is the key

- Can implement the algorithm different ways
  - **bottom-up**: fill in each entry of the table (in appropriate order)
  - **top-down**: implement formula recursively, but use a hash table to make sure that each sub-problem is solved only once
  - use whichever is easier for you
Dynamic Programming

> Can also compute actual solutions rather than just their values
> BUT that may require substantially more space
>   - space is often the bottleneck with these algorithms

> Alternatively, compute the solution from the optimal values

> Can often reduce space considerably
>   - may only need one previous row or column
>   - to get solution, use divide & conquer
>     > track the mid-point of the optimal solution along with opt value
Dynamic Programming

> Most broadly useful of these techniques
  – if you’re going to be an expert in just one, choose this one

> Most likely way to show that a problem that appears impossible is efficiently solvable

> Shortest path algorithms are also of relevance to next topic...
Network Flows

> Not all problems are solved in terms of sub-problems

> Most important example of that are network flow problems...
Network Flows

> Most general network flow problem is the following...

> **Problem**: Given a number \( k \), a graph \( G \), nodes \( s \) and \( t \), and, for each edge \( e \), bounds \( l_e \leq u_e \) on flow and a cost \( c_e \), find the *least cost* feasible flow of value \( k \).

  - flow \( f_e \) on edge \( e \) must satisfy \( f_e \leq u_e \)
  - incoming flow = outgoing flow at every node \( u \neq s, t \)
> **Problem:** Given a number $k$, a graph $G$, nodes $s$ and $t$, and, for each edge $e$, bounds $l_e \leq u_e$ on flow and a cost $c_e$, find the *least cost* feasible flow of value $k$.

- flow $f_e$ on edge $e$ must satisfy $f_e \leq u_e$
- incoming flow = outgoing flow at every node $u \neq s, t$

> **Alternative formulations**

- arbitrary demands at each individual node
- capacities on nodes in addition to edges
Tool #4: Network Flows

> **Ask:** is there a way to model the problem with bipartite matchings, disjoint paths, or cuts?
  
  – can allow multiple matchings or group restrictions
  – can support node- or edge-disjoint paths
  – can force particular edges to be used via lower bounds
  – can restrict the set of allowed subsets with infinite capacity edges
Network Flows

> Applications:
  – matching
    > covering with dominos
    > token placing
    > processor scheduling
  – disjoint paths
    > escape problem
    > airline scheduling
    > network connectivity
  – cuts
    > project selection
    > image segmentation
Network Flows

- Min-cost feasible flow generalizes two distinct problems
  - shortest path (no capacities)
  - maximum flow (no costs)

- (Overlaps with dynamic programming on shortest paths
  - problems that lie within both spheres are often shortest path problems)

- (Overlaps with matching theory for bipartite matchings
  - finding matchings in general graphs is a harder problem)
Network Flows

> Important special class of linear programming (LP) problems
  – latter are problems of minimizing a linear function of some variables subject to linear equality and inequality constraints

> **Theorem**: if all capacities and costs are integers, then there exists an *integral* min-cost flow
  – rarely easy to see when this is true for LPs

> Fractional solutions are also interesting for flows
  – example: how do we know table rounding is always possible?
Outline

- Binary Search
- Divide & Conquer
- Dynamic Programming
- Network Flows
- Branch & Bound
**Branch & Bound**

> Useful on problems that cannot be solved efficiently

> Example: NP-complete problems
  - hardest of all problems in NP

> When it looks impossible...
  - first try dynamic programming
  - then try modeling with network flows
  - then try a reduction from an NP-complete problem
    > shows your problem is NP-complete
NP-Compete Problems

> “Easiest” NP-complete problems (reduce from these):

<table>
<thead>
<tr>
<th>Packing</th>
<th>independent set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covering</td>
<td>vertex cover</td>
</tr>
<tr>
<td>Constraint Satisfaction</td>
<td>3-SAT</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Hamiltonian cycle</td>
</tr>
<tr>
<td>Partitioning</td>
<td>3D matching</td>
</tr>
<tr>
<td>Numerical</td>
<td>partition</td>
</tr>
</tbody>
</table>
Branch & Bound

> Useful on problems that cannot be solved efficiently

> Example: NP-complete problems
  – hardest of all problems in NP

> When it looks impossible...
  – first try dynamic programming
  – then try modeling with network flows
  – then try a reduction from an NP-complete problem
  – then try branch & bound
Branch & Bound

Algorithmic Approach

1. Find a convenient way to break up the solution space into pieces
   - applied recursively, this becomes a tree
   - individual solutions are the leaves of the tree

2. Find a good lower bound on value of any solution in a tree node

3. Implement a recursive search using bound to stop early
   - nodes in tree above become recursive calls
Tool #5: Branch & Bound

> **Ask:** what is the smallest subset of the constraints I could remove to make this problem efficiently solvable?
  > solving the problem with constraints removed gives a lower bound on the value of the true optimum solution
    > (upper bound in the case of a maximization problem)
    > computes the minimum of a set that includes not only all valid solutions but also invalid ones
Example: TSP (min-cost Hamiltonian cycle)

- a Hamiltonian cycle is a connected subgraph with \( \deg(u) = 2 \) for all nodes \( u \)
- bound 1: remove the \( \deg = 2 \) constraint
  - just looking for a way to connect the nodes
  - min cost solution is the minimum spanning tree
- bound 2: removing the connectedness constraint
  - just looking for \( \deg(u) = 2 \), i.e., a 2-factor
  - min cost solution is the min cost 2-factor
  - this can be modeled as a min cost flow problem
    - add node capacities with lower = upper = 1
    - split the edges to ensure only 1 direction used
Branch & Bound

> Most successful technique in practice

> You want your lower bound to be hard to compute (just not NP-hard)
  – (e.g., 2-factor requires solving a min-cost flow problem)
  – the harder the problem you are left with, the less you’ve thrown away

> Very easy to apply to integer linear programming problems
  – this is a huge class of problems
  – includes TSP and most of the other NP-complete problems that we discussed
    > that said, the more problem-specific the bound, the better
Outline

- Binary Search
- Divide & Conquer
- Dynamic Programming
- Network Flows
- Branch & Bound
- Toolkit
These are the tools that have gotten me out of almost every difficult algorithms quandary I’ve been stuck in....
Tool #1: Binary Search

> **Problem:** given inputs \((\text{input}_1, \ldots, \text{input}_k)\), compute some output
>  – i.e., compute a function \((\text{input}_1, \ldots, \text{input}_k) \rightarrow \text{output}\)

> **Ask:** would it be easier to compute \((\text{input}_1, \ldots, \text{output}) \rightarrow \text{input}_k\)？
> **Ask:** is that function monotonic?
Tool #2: Divide & Conquer

> Ask: would having the solutions to sub-problems on two halves of the data make it (truly) easier to solve?
Tool #3: Dynamic Programming

> Ask: how could the optimal solution use the last element of input?
  - for each possibility, describe the rest of the optimal solution (without the last input) as the optimal solution to a sub-problem
Tool #4: Network Flows

> Ask: is there a way to model the problem with bipartite matchings, disjoint paths, or cuts?
Tool #5: Branch & Bound

> **Ask:** what is the smallest subset of the constraints I could remove to make this problem solvable?
Questions?