Algorithm Design Patterns

- What are the common genres of algorithms? What are the approaches that are frequently used (and re-used) across different situations?
  - Why learn them? Because any basic strategy that has been widely successful in the past might be useful to know when designing a future algorithm.

- Goals:
  - Know the basic definitions and ideas behind the key algorithm design patterns.

- Optional reading: CLRS ch. 4, 5, 15, 16, 35.
Algorithm Design Patterns

- Brute Force / Exhaustive Search
- Backtracking
- Greedy
- Divide and Conquer
- Dynamic Programming
- Approximation
- Randomization
Exhaustive Search

• Example: guessing your password
  – Suppose I want to break into your FaceSpace account, but you have chosen a good password.
  – My only strategy is ____________.

• Example: prime factorization of integers
  – How to factor 576690414152577812786527 ?
  – ___________________________________________________________________
  – ___________________________________________________________________
  – ___________________________________________________________________
  – ___________________________________________________________________
  – ___________________________________________________________________
Exhaustive Search

- **Example:** guessing your password
  - Suppose I want to break into your FaceSpace account, but you have chosen a good password.
  - My only strategy is “try all strings.”

- **Example:** prime factorization of integers
  - How to factor 576690414152577812786527?
  - No very easy way to find out the answer is $(363646954537877 \cdot 1585852451)$, though it is easy to verify.
  - The best algorithms generally try lots and lots of possibilities. Cost is $O(2^k)$ for a $k$-bit integer.
Exhaustive Search

- Search for a solution by trying (“exhausting”) all possibilities, until the answer is found.
- Also known as brute force: the search is not intelligent, and is achieved simply by doing a lot of work.
- Simple but generally inefficient.
- Sometimes, nothing better is known.
  - Cryptography and hashing algorithms are deliberately designed around problems with no better (known) solution.
The 8 Queens Problem

- A classic logic puzzle: Can you place 8 Queens on a board such that none can attack any others?
The 8 Queens Problem

In chess, a Queen can move straight or on diagonals, any distance.

So the Queen here attacks many squares.
The 8 Queens Problem

We can put another queen in a location such that neither attacks the other.

Is it possible to put 8 queens on the board?
The 8 Queens Problem

There are 64 spaces, and 8 queens, so the total number of possible arrangements is:

\[
\binom{64}{8} \approx 1.78 \times 10^{14}
\]

You could spend **years** trying to solve this problem by brute force.
The 8 Queens Problem

This problem was solved in 1850.

How?
Algorithm Design Patterns

- Brute Force / Exhaustive Search
- **Backtracking**
- Greedy
- Divide and Conquer
- Dynamic Programming
- Approximation
- Randomization
Backtracking

- We can sometimes add some intelligence to exhaustive search, to waste less time.
- If a chain of decisions led to a bad answer, we can retract some of those decisions and try for better (rather than starting over from scratch).
Backtracking

In Backtracking, you solve a problem by taking small steps, and updating the problem state as you go.
Backtracking

In the 8 Queens problem, we mark off which squares in the board are attacked.
Backtracking

We now only consider squares which are not yet attacked as possible locations for another Queen.
We now only consider squares which are not yet attacked as possible locations for another Queen.
Backtracking

We now only consider squares which are not yet attacked as possible locations for another Queen.
We now only consider squares which are not yet attacked as possible locations for another Queen.
We now only consider squares which are not yet attacked as possible locations for another Queen.
This attempt to find a solution failed because we can only fit 7 queens on the board.
A Brute Force algorithm would start over from scratch, perhaps moving a single piece one space to the side. This would require roughly 100 new calculations (6 or 7 queens times a dozen attacks each) to see if the new configuration worked.
A Backtracking algorithm goes back to the previous state, and tries another alternative.
We rewind to the state before our last choice.

We mark our first choice as wrong...
We rewind to the state before our last choice.

We mark our first choice as wrong... ...and try something else.

This one doesn't work, either...but checking it was far cheaper.

We rewind again.
We see that we have no choices left to explore at this level.

So we rewind even further...
This was our state two steps back.

We mark off the choice that we have already explored...
Backtracking

This was our state **two** steps back.

We mark off the choice that we have already explored... ...and try something else.
Backtracking

Note that we don't have an X to tell us “don't try this again;” that only applied to that one choice point.

We are in a new state, so we can consider this as a possible choice again.
Another failure; we'll backtrack again.
You may have noticed that we got to the same state twice, by two different paths.

A small modification could make this impossible.

**Class Discussion:** How could you prevent duplicates?
Backtracking

- **Backtracking** is *systematic search that will discard and update past unsuccessful decisions, remembering what has been tried.*

- Think of this as exploring a huge tree, using a variation on DFS. Each vertex is a state.
Algorithm Design Patterns

- Brute Force / Exhaustive Search
- Backtracking
- **Greedy**
- Divide and Conquer
- Dynamic Programming
- Approximation
- Randomization
Greedy

- A greedy algorithm constructs its solution by a series of “locally optimal” choices.
  - “Local” choices are based on limited information; a greedy approach picks the one that seems best.
  - No backtracking.
  - Often applied to optimization problems.
- Hopefully, that leads to a globally good solution.
- Examples we have seen so far:
  - Prim's algorithm, Dijkstra's algorithm
Greed: not always a good strategy

It worked out OK for Prim, but not for this frog. (Moral: it *can* lead to a global optimum, but you have to prove it.)
Other Greedy Algorithms

• Example 1: making change (as an optimization problem)
  - Want to give $X$ cents of change to a customer while using as few coins as possible.
    • First go into “quarters mode” -- give back as many quarters as possible, without exceeding $X$.
    • Same for dimes, then nickels, then pennies.
  - Does a greedy strategy minimize the total number of coins?

• Example 2: encoding an integer in binary
Tradeoffs

• Most greedy algorithms are fast.
  – Limited number of steps to a solution.

• Many problems don't allow for greedy solutions: locally greedy choices do not lead to global optimums.
  – “Trapped in a local maximum.”

• Sometimes, a greedy algorithm makes a good heuristic -- a good approximate solution.
Algorithm Design Patterns

- Brute Force / Exhaustive Search
- Backtracking
- Greedy
- Divide and Conquer
- Dynamic Programming
- Approximation
- Randomization
Divide and Conquer

- Solve your problem by splitting the input into two or more pieces (smaller problems, not overlapping).
- Recursively solve each smaller problem.
- Merge those smaller solutions into an overall solution.
- Examples we have already seen?
Divide and Conquer

- **Solve your problem by splitting the input into two or more pieces (smaller problems, not overlapping).**
- **Recursively solve each smaller problem.**
- **Merge those smaller solutions into an overall solution.**

- **Examples already seen:**
  - Merge sort, Quicksort

- **Time complexity given by the Master Theorem.**
Algorithm Design Patterns

- Brute Force / Exhaustive Search
- Backtracking
- Greedy
- Divide and Conquer
- **Dynamic Programming**
- Approximation
- Randomization
Dynamic Programming

- **Dynamic Programming** is a technique also based on solving subproblems and merging the solutions, but it applies to situations with “overlapping” and repeated subproblems.

- We store solutions to subproblems in a table, and consult the table when we need to re-use a previous solution.

- More advanced than Divide-and-Conquer because the problems overlap and are reused.

- Name is traditional but misleading.
Dynamic Programming

• Tiny Example: Fibonacci Numbers
• What is wrong with the following algorithm?

```python
fibonacci(int n):
    if n < 0
        throw some exception
    if n == 0
        return 0
    if n == 1
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```
Recursive Fibonacci

Answer: it takes exponential time, but for no good reason.
Dynamic Programming Fibonacci

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Programming Fibonacci

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
## Dynamic Programming Fibonacci

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Programming Fibonacci

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Programming Fibonacci

cache[] = {0, 1}

fibonacci(int n):
    if n < 0
        throw some exception

    if n+1 > cache.length
        extend the array
        for i in old length..new length
            cache[i] = cache[i-1] + cache[i-2]

    return cache[i]
Algorithm Design Patterns

- Brute Force / Exhaustive Search
- Backtracking
- Greedy
- Divide and Conquer
- Dynamic Programming
- Approximation
- Randomization
Approximation

• Occasionally, you can find approximate answers quickly.
  – If they are good enough, live with it.

• Sometimes based on heuristics.
  – “This often works.”

• Sometimes based on lower bounds.
  – “It's not going to get much better than this answer.”
  – Best if we can quantify how much worse you are.
  – Example: traveling salesperson problem
Approximation

- **Traveling Salesperson problem (TSP):**
  - Given a connected, undirected, weighted graph, with nonnegative weights, find a minimum-cost cycle that includes every vertex.
  - *Euclidean* TSP: same as above, but vertices are \((x,y)\) coordinates on the plane, weights are geometric distances between vertices.
  - These problems are both NP-complete: no efficient solution is known.
  - 2-approximation to ETSP using MST (i.e., result is up to twice as large as optimal) -- details in class.
Algorithm Design Patterns

- Brute Force / Exhaustive Search
- Backtracking
- Greedy
- Divide and Conquer
- Dynamic Programming
- Approximation
- Randomization
Randomization

• A **Randomized** algorithm will make some decisions randomly.
  – In some cases, will find a good (or acceptable) answer quickly
  – Has a chance of a terrible result
  – Ideally, the average performance is good, the chance of terrible performance is statistically tiny, and the code is simple.
  – The analysis is often not so simple.

• Examples?
Randomization

• A **randomized** algorithm will make some of its decisions using a random number generator.
  – Has a chance of a terrible result.
  – Ideally, the average performance is good, the chance of terrible performance is statistically tiny, and the code is simple.
  – The analysis is often not so simple.

• Examples:
  – treap
  – randomly-chosen quicksort pivot