Algorithm Strategies

- An **algorithm strategy** is a high-level plan how to solve a problem.
  - Not an **algorithm**!

- There are many common strategies.
- Which one is best? Depends on the situation.
  - Some are fast
  - Some are easy to code
  - Different problems require different solutions
Topic 16: Algorithm Strategies

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

- Brute Force algorithms search for a solution by trying all possibilities until an answer is found
- Generally a bad idea
- Sometimes, nothing better is known
  - Crypto & hashing algorithms are often based on problems with no better known solution
Example:
- What is the first prime factor of an integer $x$?

- The best algorithms generally try lots and lots of possibilities

- There are ways to reduce the number, but not enough...

- Generally, the cost is $O(2^b)$, where $b$ is the number of bits.
When Brute Force is OK

- Sometimes, Brute Force is “good enough”
  - What is the maximum value in an unsorted array?
  - What is the sum of values in an array?
  - How many even numbers are there in a list?
- But often, if you plan ahead, there's a faster way
  - Sort the values
  - Keep a running total
  - Partition evens & odds
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:

\[ \frac{64!}{8!} = 1.78 \times 10^{14} \]

You could spend \textbf{years} trying to solve this problem by brute force.
The 8 Queens Problem

This problem was solved in 1850.

How?
Topic 16: Algorithm Strategies

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

- **Backtracking** is Brute Force with a twist: you remember things that you've done on the way to a solution.
In Backtracking, you solve a problem by taking small steps, and updating the problem state as you go.
In the 8 Queens problem, we mark off which squares in the board are attacked.
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 x 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.
Backtracking

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

So we rewind even further...
Backtracking

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.
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?
Topic 16: Algorithm Strategies

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

• A **Greedy** algorithm:
  - Makes small choices
  - Never needs to second-guess or backtrack

**Examples:**
- Prim's algorithm
- Dijkstra's algorithm
Brute Force/Backtracking/Greedy

- **Brute Force**
  - Try all possibilities

- **Backtracking**
  - Try all possible paths through the *solution space* – DFS

- **Greedy**
  - Make a decision, never reconsider it

[https://www.youtube.com/watch?v=5VwpuuaJS4g](https://www.youtube.com/watch?v=5VwpuuaJS4g)
Other Greedy Algorithms

- Making change
- Converting integer to binary
Tradeoffs

• Most greedy algorithms are fast
  - Limited # of steps to a solution

• Many algorithms don't allow for perfect greedy algorithms (mistakes are always possible)

• Sometimes, a greedy algorithm using a **heuristic** is a good (approximate) solution
  - Not perfect, but close
  - Very fast
Topic 16: Algorithm Strategies

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

- We've seen **Divide and Conquer** algorithms
  - Split the input data into two or more pieces
  - Recurse into each piece
  - Join them together

**Examples:**
- Merge Sort
- Quicksort
- Bucket Sort
Problems with Divide and Conquer

- Divide and Conquer requires that it's easy to split the data
  - Split it in half (Merge Sort / Quicksort)
  - Partition based on a simple key (Bucket Sort)

- But what if there's no obvious split?
Topic 16: Algorithm Strategies

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

- **Dynamic Programming** is more advanced than Divide-and-Conquer because
  - We don't have to split the data set early
  - We remember answers, and can re-use them
Dynamic Programming

- Classic Example: Fibonacci Numbers
  - Basic recursive algorithm is $O(2^n)$
  - Dynamic programming is $O(n)$

```python
def fibonacci(int n):
    if n < 0
        throw exception
    if n == 0
        return 0
    if n == 1
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```
Recursive Fibonacci
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></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></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></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>3</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>4</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>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

<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

<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>3</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

cache[] = {0, 1}

fibonacci(int n):
    if n < 0
        throw 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]
Dynamic Programming

• The “all to all” shortest path problem:
  “Given a graph, what is the shortest path from each node to every other node?”
Dynamic Programming

- Dynamic Programming algorithms:
  - Do **not** split the problem at the top-level (like D&C)
  - Instead, find small answers anywhere
  - Record the answers, re-use them later

- Best when we might consider the same subproblem many times
  - Can sometimes turn $O(2^n)$ algorithms into $O(n^k)$!
Topic 16: Algorithm Strategies

- 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"
Approximation

Example:
- Finding the shortest path through a real world transportation network
Approximation

Example:

- Finding the shortest path through a real world transportation network

Physical limitations give you good clues about how to find a path.
Example:

- Finding the shortest path through a real world transportation network

Say you're looking to find a path between the red dots.

A good **heuristic** would be to use northbound lines.

Using an eastbound lines will rarely give a good solution.
Approximation

Example:

- Finding the shortest path through a real world transportation network

The lower bound on the distance traveled is the distance “as the crow flies.”

Any solution that is within 20% of that number is pretty good.

It may not be worth the trouble to find a better one.
Topic 16: Algorithm Strategies

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

- A **Randomized** algorithm will make decisions randomly.
  - In some cases, will find a good (or acceptable) answer quickly
  - Has a chance of a terrible result

**Example:**
- Use a random algorithm to choose a pivot in Quicksort
Randomization

Another Example:
- Find the largest number in an input set

Proposed Algorithm:
- Randomly choose a sequence of numbers
- Keep track of the largest one you've seen
- Keep track of the number of times you've found a “new best”
- Stop when you've found 10 “new best”
Randomization

Proposed Algorithm:
- Randomly choose a sequence of numbers
- Keep track of the largest one you've seen
- Keep track of the number of times you've found a “new best”
- Stop when you've found 10 “new best”

Analysis (very rough!):
The first number you choose could be anything. On average, though, it's going to be in the middle.

Thus, you have about a 50% chance of the second number being better. On average, the second number will be about in the midpoint of the upper half – that is, larger than about 75% of all the values in the set.
Randomization

Proposed Algorithm:
- Randomly choose a sequence of numbers
- Keep track of the largest one you've seen
- Keep track of the number of times you've found a “new best”
- Stop when you've found 10 “new best”

Analysis (very rough!):
Each time you find a new number, it will be (on average) about half-way between the current number and the max.

Thus, after 10 “new best” numbers, you should be in the top .1% of all numbers in the set. That's good enough.
Randomization

Proposed Algorithm:
- Randomly choose a sequence of numbers
- Keep track of the largest one you've seen
- Keep track of the number of times you've found a “new best”
- Stop when you've found 10 “new best”

Analysis (very rough!):
Of course, each time you find a new, better number, it becomes harder to find better ones...so it will take you (on average) 1024 tries to find 10 “new best.”
Topic 16: Algorithm Strategies

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

Summary