Administrivia
Homework #9 is due Friday, Dec. 9.
Homework #10 is due Tuesday, Dec. 20.
Midterm #2 is Monday, Dec. 19 from 7–10 p.m.
Warmup Question
Question #1

x = 'ABCD'
z = 'XYZ'

for a in itertools.product( x,y ):
    print( ' '.join( a ) )

Which of the following is *not* printed?

A 'A X'
B 'B D'
C 'C X'
D 'D Z'
Question #1

```python
x = 'ABCD'
z = 'XYZ'

for a in itertools.product( x, y ):
    print( ' '.join( a ) )
```

Which of the following is *not* printed?

A  'A X'
B  'B D' ★
C  'C X'
D  'D Z'

Warmup Question
Brute-Force Search
Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.
Brute-force search

- Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

<table>
<thead>
<tr>
<th>Characters</th>
<th>Search Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86</td>
</tr>
<tr>
<td>2</td>
<td>$86^2 = 7396$</td>
</tr>
</tbody>
</table>
Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

<table>
<thead>
<tr>
<th>Characters</th>
<th>Search Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86</td>
</tr>
<tr>
<td>2</td>
<td>$86^2 = 7396$</td>
</tr>
<tr>
<td>3</td>
<td>$86^3 = 636056$</td>
</tr>
<tr>
<td>4</td>
<td>$86^4 = 54700816$</td>
</tr>
<tr>
<td>5</td>
<td>$86^5 = 4704270176$</td>
</tr>
</tbody>
</table>
Brute-force search

Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

<table>
<thead>
<tr>
<th>Characters</th>
<th>Search Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86</td>
</tr>
<tr>
<td>2</td>
<td>$86^2 = 7396$</td>
</tr>
<tr>
<td>3</td>
<td>$86^3 = 636056$</td>
</tr>
<tr>
<td>4</td>
<td>$86^4 = 54700816$</td>
</tr>
<tr>
<td>5</td>
<td>$86^5 = 4704270176$</td>
</tr>
<tr>
<td>10</td>
<td>$86^{10} = 2.2 \times 10^{19}$</td>
</tr>
<tr>
<td>20</td>
<td>$86^{20} = 4.9 \times 10^{38}$</td>
</tr>
</tbody>
</table>
Brute-force search

If Python can try a password attempt every $1 \times 10^{-7}$ s, how long does it take to crack a password of length $n$?

<table>
<thead>
<tr>
<th>Characters</th>
<th>Search Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86</td>
<td>$8.6 \times 10^{-6}$ s</td>
</tr>
<tr>
<td>2</td>
<td>7396</td>
<td>$7.4 \times 10^{-4}$ s</td>
</tr>
<tr>
<td>3</td>
<td>636 056</td>
<td>$6.4 \times 10^{-2}$ s</td>
</tr>
<tr>
<td>4</td>
<td>54 700 816</td>
<td>5.4 s</td>
</tr>
<tr>
<td>5</td>
<td>4 704 270 176</td>
<td>470.4 s</td>
</tr>
</tbody>
</table>
**Brute-force search**

- If Python can try a password attempt every $1 \times 10^{-7}$ s, how long does it take to crack a password of length $n$?

<table>
<thead>
<tr>
<th>Characters</th>
<th>Search Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86</td>
<td>$8.6 \times 10^{-6}$ s</td>
</tr>
<tr>
<td>2</td>
<td>7396</td>
<td>$7.4 \times 10^{-4}$ s</td>
</tr>
<tr>
<td>3</td>
<td>636,056</td>
<td>$6.4 \times 10^{-2}$ s</td>
</tr>
<tr>
<td>4</td>
<td>54,700,816</td>
<td>5.4 s</td>
</tr>
<tr>
<td>5</td>
<td>4,704,270,176</td>
<td>470.4 s</td>
</tr>
<tr>
<td>10</td>
<td>$2.2 \times 10^{19}$</td>
<td>$1.9 \times 10^{14}$ s</td>
</tr>
</tbody>
</table>
Brute-force search

- If Python can try a password attempt every $1 \times 10^{-7}$ s, how long does it take to crack a password of length $n$?

<table>
<thead>
<tr>
<th>Characters</th>
<th>Search Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86</td>
<td>$8.6 \times 10^{-6}$ s</td>
</tr>
<tr>
<td>2</td>
<td>7,396</td>
<td>$7.4 \times 10^{-4}$ s</td>
</tr>
<tr>
<td>3</td>
<td>636,056</td>
<td>$6.4 \times 10^{-2}$ s</td>
</tr>
<tr>
<td>4</td>
<td>54,700,816</td>
<td>5.4 s</td>
</tr>
<tr>
<td>5</td>
<td>4,704,270,176</td>
<td>470.4 s</td>
</tr>
<tr>
<td>10</td>
<td>$2.2 \times 10^{19}$</td>
<td>$1.9 \times 10^{14}$ s</td>
</tr>
<tr>
<td>20</td>
<td>$4.9 \times 10^{38}$</td>
<td>$4.9 \times 10^{31}$ s</td>
</tr>
</tbody>
</table>
Heuristic Optimization
In many cases, a “good-enough” solution is fine.
In many cases, a “good-enough” solution is fine.
If we have a figure of *relative* merit, we can classify candidate solutions by how good they are.
In many cases, a “good-enough” solution is fine. If we have a figure of *relative* merit, we can classify candidate solutions by how good they are. Heuristic algorithms don’t guarantee the ‘best’ solution, but are often adequate (and the only choice!).
Hill-climbing algorithm

- **Strategy:** Always selecting neighboring candidate solution which improves on this one.
Hill-climbing algorithm

- **Strategy:** Always selecting neighboring candidate solution which improves on this one.
- **Analogy:** Trying to find the highest hill by only taking a step uphill from where you are.

- Pitfall: Finding a local optimum instead of the global optimum.
Hill-climbing algorithm

- **Strategy:** Always selecting neighboring candidate solution which improves on this one.
- **Analogy:** Trying to find the highest hill by only taking a step uphill from where you are.
- **Pitfall:** Finding a *local* optimum instead of the global optimum.
Steepest ascent algorithm

- **Strategy:** Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.

- *Analogy:* Trying to find the highest hill by always taking the steepest step uphill from where you are.

- *Pitfall:* Finding a local optimum instead of the global optimum.
**Steepest ascent algorithm**

- **Strategy:** Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.

- **Analogy:** Trying to find the highest hill by always taking the steepest step uphill from where you are.

**Pitfall:** Finding a local optimum instead of the global optimum.
**Steepest ascent algorithm**

- **Strategy:** Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.

- **Analogy:** Trying to find the highest hill by always taking the *steepest* step uphill from where you are.

- **Pitfall:** Finding a *local* optimum instead of the global optimum.
Random sampling

Strategy: Choosing at random a candidate solution (sometimes within a constrained space).
Random sampling

- **Strategy:** Choosing at random a candidate solution (sometimes within a constrained space).
- **Analogy:** Picking random heights in the region of a hill, accepting the tallest as the highest.
Random sampling

- **Strategy:** Choosing at random a candidate solution (sometimes within a constrained space).
- **Analogy:** Picking random heights in the region of a hill, accepting the tallest as the highest.
- **Pitfall:** Without good constraints, missing the optimum value.
Random walk

- **Strategy:** Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.
Random walk

- **Strategy**: Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.
- **Analogy**: Taking random steps near a hill, but maybe not taking the step if it’s worse.

** Pitfall**: Converging slowly, can still miss best candidate solution. **BUT**: has a way from getting stuck in local optima.
Random walk

- **Strategy:** Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.
- **Analogy:** Taking random steps near a hill, but maybe not taking the step if it’s worse.
- **Pitfall:** Converging slowly, can still miss best candidate solution. BUT: has a way from getting stuck in local optima.
We require:
- A problem with relative solution assessment
- An algorithm to assess solutions
- The password cracking didn’t have the former.
- Let’s revisit the bag-packing algorithm.
Our comparative strategies:

- Brute-force (last lecture)
- Hill-climbing
Our comparative strategies:

- Brute-force (last lecture)
- Hill-climbing
  - Select heaviest item, then add next heaviest, etc.
Our comparative strategies:
- Brute-force (last lecture)
- Hill-climbing
  - Select heaviest item, then add next heaviest, etc.
  - Select most valuable item, then add next most valuable item, etc.
Our comparative strategies:

- Brute-force (last lecture)
- Hill-climbing
  - Select heaviest item, then add next heaviest, etc.
  - Select most valuable item, then add next most valuable item, etc.
- Random sampling
Our comparative strategies:

- Brute-force (last lecture)
- Hill-climbing
  - Select heaviest item, then add next heaviest, etc.
  - Select most valuable item, then add next most valuable item, etc.
- Random sampling
- Random walk: sample randomly, then iteratively allow change
import numpy as np
import matplotlib.pyplot as plt
import itertools

n = 10
items = list(range(n))
weights = np.random.uniform(size=(n,)) * 50
values = np.random.uniform(size=(n,)) * 100
```python
def f( wts, vals ):
    total_weight = 0
    total_value = 0

    for i in range( len( wts ) ):
        total_weight += wts[ i ]
        total_value += vals[ i ]

    if total_weight >= 50:
        return 0
    else:
        return total_value
```
max_value = 0.0
max_set = None
lists = []
for i in range(n):
    for set in itertools.combinations(items, i):
        wts = []
        vals = []
        for item in set:
            wts.append(weights[item])
            vals.append(values[item])
        value = f(wts, vals)
        lists.append((wts, value))
        if value > 0:
            print(value, wts)
        if value > max_value:
            max_value = value
            max_set = set
array = np.array( lists )
plt.plot( array[:,1], 'b.' )
plt.xlim( ( 0, len(lists) ) )
plt.show()
import itertools

max_value = 0.0
max_set = None
for i in range(n):
    for set in itertools.combinations(items, i):
        wts = []
        vals = []
        for item in set:
            wts.append(weights[item])
            vals.append(values[item])
        value = f(wts, vals)
        if value > max_value:
            max_value = value
            max_set = set
Hill-climbing search

max_wt = 50.0

wts_orig = wts[ : ]
vals_orig = vals[ : ]

best_vals = [ ]
best_wts = [ ]
best_vals.append( max( vals ) )
best_wts.append( wts[ vals.index( max( vals ) ) ] )
wts.remove( wts[ vals.index( max( vals ) ) ] )
vals.remove( max( vals ) )
Hill-climbing search

while sum(best_wts) + wts[vals.index(max(vals))] < max_wt:
    best_vals.append(max(vals))
    best_wts.append(wts[vals.index(max(vals))])
    wts.remove(wts[vals.index(max(vals))])
    vals.remove(max(vals))

wts = wts_orig[:]
vals = vals_orig[:]

Heuristic Optimization
# try a configuration at random
# alter it at random with small likelihood of getting worse
for t in range(1000):
    # two possible moves: adding or removing
    if f(next_wts,next_vals) > f(trial_wts,trial_vals):
        # if improvement, accept the change
    else:
        # if no improvement, *maybe* accept the change
    # if all-time best, track it
# (see random-walk.py)
In order to compare algorithms, we need a way to measure code run time (called “wallclock time”).
In order to compare algorithms, we need a way to measure code run time (called “wallclock time”). The `timeit` module provides three ways to time your code:
In order to compare algorithms, we need a way to measure code run time (called “wallclock time”).

The `timeit` module provides three ways to time your code:

- **Interpreter:**
  ```python
  timeit.timeit('func(n)', number=10000)
  ```

- **Command line:**
  ```bash
  python3 -m timeit 'code'
  ```

- **Notebook:**
  ```python
  %timeit func(n)
  ```

These run your code many times and return an average time to completion.
In order to compare algorithms, we need a way to measure code run time (called “wallclock time”).

The `timeit` module provides three ways to time your code:

- **Interpreter:** `timeit.timeit('func(n)', number=10000)`
- **Command line:** `python3 -m timeit 'code'`
In order to compare algorithms, we need a way to measure code run time (called “wallclock time”). The `timeit` module provides three ways to time your code:

- **Interpreter**: `timeit.timeit('func( n )', number=10000)`
- **Command line**: `python3 -m timeit 'code'
- **Notebook**: `%timeit func( n ) (this is easiest)`
In order to compare algorithms, we need a way to measure code run time (called “wallclock time”).

The `timeit` module provides three ways to time your code:
- Interpreter: `timeit.timeit('func(n)', number=10000)`
- Command line: `python3 -m timeit 'code'`
- Notebook: `%timeit func(n)` (this is easiest)

These run your code many times and return an average time to completion.
Fibonacci sequence

\[ F_n = F_{n-1} + F_{n-2} \quad F_1 = F_2 = 1 \]

1 1 2 3 5 8 13 21 34 55 …
Fibonacci sequence

\[ F_n = F_{n-1} + F_{n-2} \quad F_1 = F_2 = 1 \]

1 1 2 3 5 8 13 21 34 55 ...

\[ F_n = \frac{\left( \frac{1+\sqrt{5}}{2} \right)^n + \left( \frac{2}{1+\sqrt{5}} \right)^n}{\sqrt{5} + \frac{1}{2}} \]
def fib_a( n ):
    sqrt_5 = 5**0.5;
    p = ( 1 + sqrt_5 ) / 2;
    q = 1 / p;
    return int( (p**n + q**n) / sqrt_5 + 0.5 )
def fib_r( n ):
    if n == 1 or n == 2:
        return 1
    else:
        return fib_r( n-1 ) + fib_r( n-2 )
Comparison

```python
%timeit fib_a( 12 )
%timeit fib_r( 12 )
```

On my machine, `fib_a` is 55 faster than `fib_r` for `n = 12`. (Will this performance get better or worse for larger `n`?)
%timeit fib_a( 12 )
%timeit fib_r( 12 )

▷ On my machine, fib_a is 55 × faster than fib_r for n = 12. (Will this performance get better or worse for larger n?)
Comparing Results
arrays don’t play nicely with comparisons:

```python
one = np.ones((5,))
if one == 1:
    print('setup correct')
```

ValueError: The truth value of an array with more than one element is ambiguous. Which element is compared? It’s ambiguous.
Comparing results

- arrays don’t play nicely with comparisons:
  one = np.ones((5,))
  if one == 1:
      print('setup correct')
  ValueError: The truth value of an array with more than one element is ambiguous.
arrays don’t play nicely with comparisons:

```python
one = np.ones((5,))
if one == 1:
    print('setup correct')
ValueError: The truth value of an array with more than one element is ambiguous.
```

Which element is compared? It’s ambiguous.
arrays have the built-in methods any and all:

```python
one = np.ones((5,))
if one.all() == 1:
    print('setup correct')
```
arrays have the built-in methods any and all:

```python
one = np.ones((5,))
if one.all() == 1:
    print( 'setup correct' )

domain = np.linspace(0,10,11)
if domain.any() == 1:
    print( 'setup contains one' )
```