Numerical Python

optimization

CS101 Lecture #20

2016-12-07

Administrivia

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

```
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'`

Brute-Force Search

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.

Characters	Search Space
1	86
2	$86^2 = 7\,396$

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.

Characters	Search Space
1	86
2	$86^2 = 7\,396$
3	$86^3 = 636\,056$
4	$86^4 = 54\,700\,816$
5	$86^5 = 4\,704\,270\,176$

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.

Characters	Search Space
1	86
2	$86^2 = 7\,396$
3	$86^3 = 636\,056$
4	$86^4 = 54\,700\,816$
5	$86^5 = 4\,704\,270\,176$
10	$86^{10} = 2.2 \times 10^{19}$
20	$86^{20} = 4.9 \times 10^{38}$

Brute-force search

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

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	$7\,396$	7.4×10^{-4} s
3	$636\,056$	6.4×10^{-2} s
4	$54\,700\,816$	5.4 s
5	$4\,704\,270\,176$	470.4 s

Brute-force search

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

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	$7\,396$	7.4×10^{-4} s
3	$636\,056$	6.4×10^{-2} s
4	$54\,700\,816$	5.4 s
5	$4\,704\,270\,176$	470.4 s
10	2.2×10^{19}	1.9×10^{14} s

Brute-force search

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

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	7 396	7.4×10^{-4} s
3	636 056	6.4×10^{-2} s
4	54 700 816	5.4 s
5	4 704 270 176	470.4 s
10	2.2×10^{19}	1.9×10^{14} s
20	4.9×10^{38}	4.9×10^{31} s

Heuristic Optimization

Heuristic optimization

- In many cases, a “good-enough” solution is fine.

Heuristic optimization

- 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 optimization

- 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

- **S**trategy: Always selecting neighboring candidate solution which improves on this one.

Hill-climbing algorithm

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

Hill-climbing algorithm

- **S**trategy: Always selecting neighboring candidate solution which improves on this one.
- **A**nalogy: Trying to find the highest hill by only taking a step uphill from where you are.
- **P**itfall: Finding a *local* optimum instead of the global optimum.

Steepest ascent algorithm

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

Steepest ascent algorithm

- **S**trategy: Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.
- **A**nalogy: Trying to find the highest hill by always taking the *steepest* step uphill from where you are.

Steepest ascent algorithm

- **S**trategy: Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.
- **A**nalogy: Trying to find the highest hill by always taking the *steepest* step uphill from where you are.
- **P**itfall: Finding a *local* optimum instead of the global optimum.

Random sampling

- **S**trategy: Choosing at random a candidate solution (sometimes within a constrained space).

Random sampling

- **S**trategy: Choosing at random a candidate solution (sometimes within a constrained space).
- **A**nalogy: Picking random heights in the region of a hill, accepting the tallest as the highest.

Random sampling

- **S**trategy: Choosing at random a candidate solution (sometimes within a constrained space).
- **A**nalogy: Picking random heights in the region of a hill, accepting the tallest as the highest.
- **P**itfall: Without good constraints, missing the optimum value.

Random walk

- **S**trategy: Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.

Random walk

- **S**trategy: Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.
- **A**nalogy: Taking random steps near a hill, but maybe not taking the step if it's worse.

Random walk

- **S**trategy: Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.
- **A**nalogy: Taking random steps near a hill, but maybe not taking the step if it's worse.
- **P**itfall: Converging slowly, can still miss best candidate solution. BUT: has a way from getting stuck in local optima.

Example

- 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.

Example

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

Example

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

Example

- 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.

Example

- 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

Example

- 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

Setup

```
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
```

Setup

```
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
```

Tracking cases

```
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
```

Tracking cases

```
array = np.array( lists )
plt.plot( array[:,1], 'b.' )
plt.xlim( ( 0, len(lists) ) )
plt.show()
```

Brute-force search

```
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[ : ]
```

Random walk

```
# 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)
```

Code Performance

Code performance

- In order to compare algorithms, we need a way to measure code run time (called “wallclock time”).

Code performance

- 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:

Code performance

- 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)`

Code performance

- 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'`

Code performance

- 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)

Code performance

- 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

$$F_n = F_{n-1} + F_{n-2} \qquad\qquad F_1 = F_2 = 1$$

$$1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ 34\ 55\ \ldots$$

Fibonacci sequence

$$F_n = F_{n-1} + F_{n-2} \qquad\qquad F_1 = F_2 = 1$$

$$1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ 34\ 55\ \ldots$$

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{2}{1+\sqrt{5}}\right)^n}{\sqrt{5}+\frac{1}{2}}$$

Analytical Fibonacci

```
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 )
```

Recursive Fibonacci

```
def fib_r( n ):
    if n == 1 or n == 2:
        return 1
    else:
        return fib_r( n-1 ) + fib_r( n-2 )
```

Comparison

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

Comparison

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

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

Comparing Results

Comparing results

- arrays don't play nicely with comparisons:

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

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.
```

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.
```

- Which element is compared? It's ambiguous.

Comparing results

- `arrays` have the built-in methods `any` and `all`:

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

Comparing results

- arrays have the built-in methods `any` and `all`:

```
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' )
```