Python Intermediate Programming

Unit 4: Algorithmic Study

CHAPTER 12: EMPIRICAL EFFICIENCY

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- In the two previous lectures we learned about complexity classes and how to analyze algorithms (mostly Python statements/functions) to find their complexity classes. These analyses were mostly done by just looking at code (aka “static analysis”) and were independent of any technology: i.e., independent of which Python interpreters we used and the speed of the computers running our code.

Objectives

- We also learned that given the complexity class of a runnable Python function, we can approximate its running time by `T(N) = c*complexity_class(N)`, where `complexity_class(N)` is its complexity class: e.g., `N`, `N Log N`, `N**2`, etc. We can then run this function on a reasonably-sized problem (with `N` not too small, so the discarded lower-order terms are small enough to really ignore) and measure the amount of time it takes `(T)`. Finally, we can solve for the constant in the time equation: `c = T(N)/complexity(N)` by plugging in `T(N)` and `complexity(N)`.
- Then, we can use the formula `T(N) = c*complexity_class(N)`, with the computed c, to approximate the amount of time this function requires to solve a problem of any size N. Such analysis (by running code) is called "dynamic analysis". We also saw that we could approximate the complexity class by running the code on input sizes that double and then plot the results, looking for a match against standard doubling signatures.

Objectives

- In the first part of this lecture, we will examine how to time functions on the computer (rather than using an external timer) and we will write some Python code to help automate this task. Given such tools, and the ability to chart the time required for various-sized problems, we can also use this data to infer the complexity class of a function without ever seeing its code. Yet we can still develop a formula `T(N)` to approximate the amount of time this function requires to solve a problem of any size `N`, without even looking at the code.
- Generally, in this lecture we will explore using the computer (dynamic analysis) to better help us understand the behavior of algorithms that might be too complex for us to understand by static analysis (we might not even have the algorithm in the form of code, so we cannot examine it, only run it).

Objectives

- In the second section, we will switch scale and use a Profiling module/tool named `cProfile` (and its associated module `pstats`) to run programs and determine which functions are consuming the most time. Once we know this, we will an attempt to improve the performance of the program by optimizing only those functions that are taking significant time.
- Finally, in the third section we will explore how Python uses **HASHING** in `sets`, `frozen sets`, and `dicts` to achieve a constant time complexity class, `O(1)` for many operations. We will close the loop by using dynamic analysis to verify this `O(1)` complexity class. Hashing is more closely studied in ICS-46.

Understanding Sorting and N Log N Algorithms

LECTURE 1

Understanding Sorting and N Log N Algorithms

- We previously discussed that the fastest algorithm for sorting a list is in the complexity class `O(N Log N)`. This is true not only for the actual algorithm used in the Python sort method (operating on lists), but for all possible algorithms. In this section, without using this knowledge, we will time the sorting function and infer its complexity class. In the process we will build a sophisticated timing tool for timing sorting and other algorithms (and use it a few times later in this lecture).
- Here are Python statements that generate a list of a million integers, shuffle them into a random order, and then sort them.

Understanding Sorting and N Log N Algorithms

```
#1_000_000 Python 3.6 allows _ in numbers
alist = [i for i in range(1000000)]
random.shuffle(alist)
alist.sort()
```

- Let's first discuss how to time this code. There is a time module in Python that supplies access to various system clocks and conversions among various time formats. I have written a Stopwatch class (in the `stopwatch.py` module, which is part of **`courselib`**) that makes clocks easier to use for timing code. The four main methods for objects constructed from Stopwatch classes are start, stop, read, and reset.

Understanding Sorting and N Log N Algorithms

- You can read the complete documentation for this class by following the "Course Library Reference" link and then clicking the Stopwatch link. The code itself (if you are interested in reading it) is accessible in Eclipse by disclosing the `python.exe` in the "PyDev Package Explorer", then disclosing "System Libs“ and then "`workspace/courselib`", and finally clicking on stopwatch.py to see the code.
- Here is a complete script that uses a Stopwatch to time only the `alist.sort()` part of the code above. Notice that it imports the **`gc`** (garbage collection) module to turn off garbage collection during the timing, so this process will not interfere with our measurements. We need to turn it back on afterwards.

Objects, Memory, and Garbage (collection)

- When we construct an object in Python, we allocate part of the computer's memory to store the object's attributes/state. The following loop will eventually consume all the memory Python can use (it runs for a few seconds on my computer). It tries to store a million factorials in a list: ultimately it raises a **MemoryError** exception.

```
from goody import irange
facts = []
answer = 1
for i in irange(1,1000000):
    answer *= i
    facts.append(answer)
```

Objects, Memory, and Garbage (collection)

- Garbage is objects constructed by Python, which can no longer be referred to. If we wrote `x = [i for i in range(1000000)]` then `x` would refer to a list object that stored a huge amount of memory. If we then wrote `x=1` the list object that x used to refer to would become garbage (because x no longer refers to this object and there are no other names that we can use to reach this object). If we had instead written

```
x = [i for i in range(1000000)]
y = [0, 1, 2, x]
x = 1
```

Objects, Memory, and Garbage (collection)

- Now, the object x formerly referred to can be referred to by `y[3]` so we can reach it from some name and therefore it is not garbage.
- Garbage collection is a way for the computer to find/reclaim memory that is garbage. Typically, Python allocates memory for objects until it finds it has no more memory to allocate; then it does garbage collection to try to find more.
- If it succeeds this process continues until it runs out of memory again, and then repeats. If it ever cannot find enough free memory to allocate an object, even after garbage collection (as in the first example) it raises an exception.

Objects, Memory, and Garbage (collection)

- Finally, there are ways to tell Python how much of the computer's memory it can use to allocate the objects it constructs. You will learn more about garbage and garbage collection in ICS-45C (using C++, a language that does not have automatic garbage collection) and ICS-46.
- A better term would be recyclables and recycling, because memory is never thrown away, but is recycled.

stopwatch.py

```python
import random,gc
from stopwatch import Stopwatch

#setup
alist = [i for i in range(1000000)]
random.shuffle(alist)
s = Stopwatch()
gc.disable()

#timing
s.start()
alist.sort()
s.stop()

#results
print('Time =',s.read())
gc.enable()
```

Performance Tool

- We would like to develop a **Performance** tool that minimizes what we have to do to time the code: a tool that also gives us interesting information about multiple timings. The tool I developed for this lecture is based on the **`timeit.py`** module supplied with Python (see secton 27.5 in the standard library documentation). First, we show an example use of the tool, then its code. Here is the entire script that uses the tool.

```
alist = [i for i in range(100000)]
p = Performance(lambda:alist.sort(),\
    lambda:random.shuffle(alist),100,'Sorting')
p.evaluate()
p.analyze()
```

Performance Tool

- It first creates a list of 100,000 numbers. Then it constructs a Performance object with 4 arguments
  1) A parameterless lambda of the code to execute and time
  2) A parameterless lambda of the setup code to execute (but not time) before each (1) lambda is called
  3) The number of times to measure the code's execution: doing step 2 followed by timing step 1
  4) A short title (printed in the analyze method)

- The actual __init__ function for Performance looks like;

```
def __init__(self,code,setup=lambda:None,times_to_measure=5,title='Generic'):
```

Performance Tool

- So, in the above call we are timing a call to `alist.sort();` before timing each call it sets up (not part of the timing) with a call to `random.shuffle(alist);` it will do 100 timings (of the setup/code); the title when information is printed is Sorting.
- Calling `p.evaluate()` does all the timings and collects the information. It returns a 2-list: a 3-tuple of the (minimum time, average time, maximum time), followed by a list of all the (100 in this case) timings. It also saves this information as part of the state of the Performance object, which is used for analysis, if we call the analyze function.

Performance Tool

- Calling `p.analyze()` prints the following result on my computer. It consists of the title; the number of timings; the average time, the minimum time, the maximum time, and the span (a simple approximation to clustering: (max-min)/avg; and a histogram of the timings (how many fall in the range .0404-.0406 bin, .0406-.0408, bin, etc.) with an 'A' at the top of the stars indicating the bin for the average time. Notice that although the span says the range of values was 5.9% of the average, we can see that most of the timings are clustered very close to the average (which itself is near the minimum time).

Sorting

Analysis of 100 timings

```
avg = 0.041   min = 0.040  max = 0.043  span = 5.9%
```

Time Ranges

```
4.04e-02<>4.06e-02[ 58.4%]|**************************************************
4.06e-02<>4.08e-02[ 20.8%]|*****************A
4.08e-02<>4.11e-02[  6.9%]|*****
4.11e-02<>4.13e-02[  5.0%]|****
4.13e-02<>4.16e-02[  5.0%]|****
4.16e-02<>4.18e-02[  0.0%]|
4.18e-02<>4.20e-02[  1.0%]|
4.20e-02<>4.23e-02[  1.0%]|
4.23e-02<>4.25e-02[  0.0%]|
4.25e-02<>4.28e-02[  1.0%]|
4.28e-02<>4.30e-02[  1.0%]|
```

Sorting

- Using this tool, I ran a series of sorting experiments doubling the length of the list to sort each time. Here was the script:

```
for i in irange(0,9)   :
    alist = [i for i in range(100000 * 2**i)]
    p = Performance(lambda : alist.sort(), \
        lambda: random.shuffle(alist),10,'Sorting')
    p.evaluate()
    p.analyze()
```

Sorting

- The data produced is summarized as follows (the raw data appears at the end of this document).

N	Time	Ratio	Predicted	%Error
100,000	0.041		0.051	23
200,000	0.088	2.1	0.107	22
400,000	0.194	2.2	0.227	17
800,000	0.440	2.3	0.478	9
1,600,000	1.004	2.3	1.004	0
3,200,000	2.291	2.3	2.105	8
6,400,000	5.058	2.2	4.406	13
12,800,000	11.380	2.2	9.201	19
25,600,000	25.185	2.2	19.182	24
51,200,000	60.692	2.4	39.992	24

Sorting

- I sorted lists from 100 thousand to 51.2 million values, doubling the length every time, whose sizes are listed in the first column. The average times (from 10 experiments each) are listed in the second column. I computed the ratio of `T(2N)/T(N)` for each N (after the first) and the ratio was always bigger than 2 by a small amount. This indicates that the complexity class is slightly higher than just `O(N)`. As we discussed, it is actually `O(N Log N)`, and this is the signature for `O(N Log N)`: slightly bigger than 2.

- Using `O(N Log N)` as the complexity class and using `N = 1,600,000` I solved for the constant in the formula `T(N) = c * N Log N` and got `3.04166E-08`, so we can approximate the time taken to sort as `T(N) = 3.042x10^-8 * N Log N`. Given this approximation, the next columns shows the times predicted for that size `N`, and the percent error between the predicted and real time (which grows as `N` gets farther away -in both directions- from `1,600,000`). The errors are not bad: even a 100% error means that we have still predicted the time within a factor of `2`, and here the worst error was about `25%`.

Sorting

- Here is the actual code for the Performance class. You will see that although the constructor specifies `times_to_measure`, we can omit/override this value when calling `evaluate()` by passing the number of times to test the code.
- Likewise with the title and the analyze method (which also allows specification of the number of bins to use in the histogram of times created).

```
import random,gc
from stopwatch import Stopwatch
from goody import irange,frange
class Performance:
    def __init__(self,code,setup=lambda:None,times_to_measure=5,title='Generic'):
        self._code              = code
        self._setup             = setup
        self._times             = times_to_measure
        self._evaluate_results = None
        self._title             = title

    def evaluate(self,times=None):
        results = []
        s = Stopwatch()
        times = times if times != None else self._times
        for i in range(times):
            self._setup()
            s.reset()
            gc.disable()
            s.start()
            self._code()
            s.stop()
            gc.enable()
            results.append(s.read())
        self._evaluate_results = [(min(results),sum(results)/times,max(results))] + [results]
        return self._evaluate_results
```

```
def analyze(self,bins=10,title=None):
       if self._evaluate_results == None:
           print('No results from calling evaluate() to analyze')
           return

       def print_histogram(bins_dict):
           count = sum(bins_dict.values())
           max_for_scale = max(bins_dict.values())

           for k,v in sorted(bins_dict.items()):
               pc = int(v/max_for_scale*50)
               extra = 'A' if k[0] <= avg < k[1] else ''
               print('{bl:.2e}<>{bh:.2e}[{count: 5.1f}%]|{stars}'.format(bl=k[0],bh=k[1],    \
                     count=v/count*100,      \
                     stars='*'*pc+extra))
```

```
(mini,avg,maxi),times = self._evaluate_results
incr = (maxi-mini)/bins
hist = {(f,f+incr):0 for f in frange(mini,maxi,incr)}
for t in times:
    for (min_t,max_t) in hist:
        if min_t<= t < max_t:
            hist[(min_t,max_t)] += 1

print(title if title != None else self._title)
print('Analysis of',len(times),'timings')
print('avg = {avg:.3f}   min = {min:.3f}  max = {max:.3f}  span = {span:.1f}%'.
      format(min=mini,avg=avg,max=maxi,span=(maxi-mini)/avg*100))
print('\n   Time Ranges    ')
print_histogram(hist)
```

Sorting

- This class is in the `performance.py` module in the empirical project folder: a download that accompanies this lecture. So, you can run your own experiments by importing this module wherever is code that you want to time.

Heights of Random Binary Search Trees: A Dynamic Analysis

LECTURE 2

Heights of Random Binary Search Trees: A Dynamic Analysis

- Let's empirically examine the heights of binary search trees constructed at random: values are added into binary search trees in random orders. We know that the maximum/worst-case height for a binary search tree with size N is N-1; the minimum/best case is a height of `(Log2 N)-1`. We will write code below to perform a specified number of experiments, each building a random binary search tree of a specified size: for each experiment we will collect the height of the tree produced and ultimately plot a histogram of all the heights.

- To run these experiments, we need to access the `TN` class and the height, add, and `add_all` functions, which are all written in the tree project folder (examined when we discussed trees). I copied that code into the `randomtrees.py` module, but we could have imported it.

Here is the code to prompt the user for the experiment and compute a histogram of all the different tree heights.

```
import prompt,random,math
from goody import irange
from collections import defaultdict

experiments = prompt.for_int('Enter # of experiments to perform')
size        = prompt.for_int('Enter size of tree for each experiment')

hist  = defaultdict(int)
alist = [i for i in range(size)]

for exp in range(experiments):
    if exp % (experiments//100) == 0:
        print('Progress: {p:d}%'.format(p =int(exp/(experiments//100))))
    random.shuffle(alist)
    hist[ height(add_all(None,alist)) ] += 1

print_histogram('Binary Search Trees',hist)
print('\nminimum possible height =',math.ceil(math.log2(size)-1),'  maximum possible height =',size-1)
```

Heights of Random Binary Search Trees

- For 10,000 experiments run on binary search trees of size 1,000, this code printed the following results (after computing for a few minutes). For a 1,000-node tree, the minimum possible height is 9 and the maximum possible height is 999. The heights recorded here are all between about 1.5 times the minimum and about 3 times the minimum (which is true for much larger random binary search trees as well; see the next analysis).

Binary Search Trees

Analysis of 10,000 experiments

```
avg = 21.0  min = 16  max = 31
  16[  0.0%]|
  17[  1.0%]|**
  18[  5.4%]|************
  19[ 14.3%]|*********************************
  20[ 21.2%]|*************************************************
  21[ 21.5%]|**************************************************A
  22[ 16.1%]|*************************************
  23[  9.8%]|**********************
  24[  5.7%]|*************
  25[  2.9%]|******
  26[  1.3%]|**
  27[  0.5%]|*
  28[  0.1%]|
  29[  0.1%]|
  30[  0.0%]|
  31[  0.0%]|
```

minimum possible height = 9   maximum possible height = 999

Note because the 16, 30, 31 bins are printed, they were not 0 (there were randomly constructed trees with those heights), although they had so few trees that their percentage (to one decimal place) is 0.

The print_histogram method called in the code above is shown below

```
def print_histogram(title,bins_dict):
    print(title)

    count = sum(bins_dict.values())
    min_bin = min(bins_dict.keys())
    max_bin = max(bins_dict.keys())
    max_for_scale = max(bins_dict.values())
    print('Analysis of {count:,} experiments'.format(count=count))

    w_sum = 0
    for i in bins_dict:
        w_sum += i*bins_dict[i]
    avg = w_sum/count

    print('\navg = {avg:.1f}  min = {min}  max = {max}\n'.format(avg=avg,min=min_bin,max=max_bin))
    for i in irange(min_bin,max_bin):
        pc = int(bins_dict[i]/max_for_scale*50)
        extra = 'A' if int(avg+.5) == i else ''
        print('{bin:4}[{count: 5.1f}%]|{stars}'.format(bin=i,count=bins_dict[i]/count*100,stars='*'*pc+extra))
```

Heights of Random Binary Search Trees

- The results below are for 10,000 experiments run on binary search trees of size 100,000. This code took about 5 hours to run. Notice too that almost all the trees are between 2 and 3 times the minimum possible tree; none are near the maximum of 99,9999.

Binary Search Trees

Analysis of 10,000 experiments

```
avg = 39.6  min = 34  max = 53
  34[  0.1%]|
  35[  0.7%]|*
  36[  4.2%]|**********
  37[ 11.1%]|****************************
  38[ 16.9%]|******************************************
  39[ 19.7%]|**************************************************
  40[ 17.1%]|*******************************************A
  41[ 12.5%]|*******************************
  42[  8.0%]|********************
  43[  4.9%]|************
  44[  2.4%]|******
  45[  1.4%]|***
  46[  0.7%]|*
  47[  0.3%]|
  48[  0.2%]|
  49[  0.1%]|
  50[  0.1%]|
  51[  0.0%]|
  52[  0.0%]|
  53[  0.0%]|
```

minimum possible height = 16   maximum possible height = 99999

All this code is in the randomtrees.py module in the empirical project folder: a download that accompanies this lecture. So, you can run your own experiments.

Profiling Programs:

LECTURE 3

Profiling Programs:

Performance Summary of all Functions at the Program Level

- Profilers are modules that run other modules and collect information about their execution: typically, information about how many times their functions are called, and the time spent inside each function: both the individual time and the cumulative time, which also includes the amount of time spent inside the functions they call. For example, in

```
def f(...):
    ...code 1
    g(...)
    ...code 2
```

Profiling Programs

- The INDIVIDUAL time for f includes the amount of time spent in code 1 and code 2; the CUMULATIVE time for f also includes the amount of time spent in function g; of course, g will have its own individual and cumulative times too.
- Such information is useful for focusing our attention on the functions taking a significant amount of time, so that we can optimize/rewrite them in the hope of significantly improving the speed of our entire program (and not wasting our time optimizing/rewriting functions that do not significantly affect their running time).

Profiling Programs

- Although the programs we wrote this quarter were sophisticated, they ran on only a small amount of data and executed quickly. The ideal program to profile is one that uses a lot of data and takes a long time to run. In my ICS-31 class, students wrote a program that performs shotgun assembly on DNA strands. The program isn't huge (only about 50 lines of code), but in the largest problem they solve, the input is 1,824 DNA fragments that are 50-100 bases long. My solution program took almost about 1.5 minutes to run, before printing the resulting 10,000 base DNA strand that it built from the fragments.
- This program is a module that defines functions followed by a script that calls these functions. To run it using the profiler module, we need to move the statements in the script into their own function (which I generically called `perform_task`). Then, we add the following import at the top, and add the following function call at the bottom to profile the function/program.

Profiling Programs

```
import cProfile
```

...all the functions in the module, including perform_task

```
cProfile.run('perform_task()')
```

- When run, the DNA assembly program performs its job and produces the correct output (and in the process, prints information into the console). Then the profiler prints the following information in the console, which shows in the top line that it profiled 234 million function calls over 107 seconds: so overall Python called 2.19 million functions/second.

Profiling Programs

- The data shown here (and all the data shown below later) is always sorted by one of the three headings. This data is sorted by the ASCII values in the strings produced by `filename:lineno` (function). The columns (no matter how they are sorted) represent

ncalls: the number of times the function was called

tottime: the total time spent in just that function, NOT INCLUDING the time spent in the other functions that it calls (although some built-in functions it calls cannot be timed separately) This as a bad name; I'd prefer individual time; but it is total time.

cumtime: the cumulative time spent in that function, INCLUDING the time spent in the other functions that it calls

Profiling Programs

- So, as illustrated above, if function f performed some computation and called function `g`, the `tottime` for `f` would NOT include the time spent in `g`, but the cumtime would include this time. So, it should always be the case that `tottime<=cummtime`.

- Examine the information shown below, but we will look at parts of it more selectively soon. Note that the sum of all the `tottime` data is the running time, but many `cumtime` data have the same value as the total running time (or close): the `<module>, exc, perform_task`, and assemble all show a cumulative time equal to the running time, because `cumtime` form them is counted in the functions they call.

233,525,220 function calls in 106.705 seconds

Ordered by: standard name

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  106.705  106.705 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 codecs.py:164(__init__)
        1    0.000    0.000    0.000    0.000 codecs.py:238(__init__)
        6    0.000    0.000    0.000    0.000 cp1252.py:18(encode)
       14    0.000    0.000    0.000    0.000 cp1252.py:22(decode)
  4671030    3.541    0.000    4.028    0.000 goody.py:17(irange)
     3627    1.311    0.000    1.768    0.000 listlib.py:16(remove)
        2    0.000    0.000    0.000    0.000 locale.py:555(getpreferredencoding)
  4671030   39.505    0.000  103.272    0.000 profilednamaker.py:10(max_overlap)
        1    0.000    0.000    0.001    0.001 profilednamaker.py:19(read_fragments)
        1    0.001    0.001    0.001    0.001 profilednamaker.py:20(<listcomp>)
     1814    1.650    0.001  104.921    0.058 profilednamaker.py:23(choose)
        1    0.010    0.010  106.699  106.699 profilednamaker.py:33(assemble)
  1658018    0.152    0.000    0.152    0.000 profilednamaker.py:42(<lambda>)
  1656204    0.155    0.000    0.155    0.000 profilednamaker.py:43(<lambda>)
        1    0.004    0.004  106.705  106.705 profilednamaker.py:49(perform_task)
        7    0.000    0.000    0.000    0.000 profilednamaker.py:55(<lambda>)
        7    0.000    0.000    0.000    0.000 profilednamaker.py:56(<lambda>)
194186759   58.607    0.000   58.607    0.000 profilednamaker.py:6(overlaps)
        2    0.000    0.000    0.000    0.000 {built-in method _getdefaultlocale}
       14    0.000    0.000    0.000    0.000 {built-in method charmap_decode}
        6    0.000    0.000    0.000    0.000 {built-in method charmap_encode}
        1    0.000    0.000  106.705  106.705 {built-in method exec}
 22001998    1.140    0.000    1.140    0.000 {built-in method len}
  4671030    0.630    0.000    0.630    0.000 {built-in method min}
        2    0.000    0.000    0.000    0.000 {built-in method open}
        3    0.000    0.000    0.000    0.000 {built-in method print}
     1813    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1824    0.000    0.000    0.000    0.000 {method 'rstrip' of 'str' objects}
        1    0.000    0.000    0.000    0.000 {method 'sort' of 'list' objects}
```

Profiling Programs

- By calling `cProfile.run('perform_task()','profile')` we direct the run function to not print its results on the console, but instead to write them into the file named 'profile' (or any other file name we want to use). Then we can use the `pstats` module, described below, to read this data file and print it in simplified forms.
- Here is a script that uses `pstats` to show just the top 10 lines of the data above, when sorted by `ncalls, cumtime, and tottime`.

Profiling Programs

```
import pstats
p = pstats.Stats('profile')
# uncomment the line below to print all the the information above
#p.strip_dirs().sort_stats(-1).print_stats()
p.sort_stats('calls').print_stats(10)
p.sort_stats('cumulative').print_stats(10)
p.sort_stats('time').print_stats(10)
```

- The three results this script prints are

    233,5252,20 function calls in 106.705 seconds

Ordered by: call count
List reduced from 31 to 10 due to restriction <10>

```
  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
194186759   58.607    0.000   58.607    0.000 profilednamaker.py:6(overlaps)
 22001998    1.140    0.000    1.140    0.000 {built-in method len}
  4671030    3.541    0.000    4.028    0.000 goody.py:17(irange)
  4671030    0.630    0.000    0.630    0.000 {built-in method min}
  4671030   39.505    0.000  103.272    0.000 profilednamaker.py:10(max_overlap)
  1658018    0.152    0.000    0.152    0.000 profilednamaker.py:42(<lambda>)
  1656204    0.155    0.000    0.155    0.000 profilednamaker.py:43(<lambda>)
     3627    1.311    0.000    1.768    0.000 listlib.py:16(remove)
     1824    0.000    0.000    0.000    0.000 {method 'rstrip' of 'str' objects}
     1814    1.650    0.001  104.921    0.058 profilednamaker.py:23(choose)
```

233,525,220 function calls in 106.705 seconds

```
Ordered by: cumulative time
  List reduced from 31 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  106.705  106.705 {built-in method exec}
        1    0.000    0.000  106.705  106.705 <string>:1(<module>)
        1    0.004    0.004  106.705  106.705 profilednamaker.py:49(perform_task)
        1    0.010    0.010  106.699  106.699 profilednamaker.py:33(assemble)
     1814    1.650    0.001  104.921    0.058 profilednamaker.py:23(choose)
  4671030   39.505    0.000  103.272    0.000 profilednamaker.py:10(max_overlap)
194186759   58.607    0.000   58.607    0.000 profilednamaker.py:6(overlaps)
  4671030    3.541    0.000    4.028    0.000 goody.py:17(irange)
     3627    1.311    0.000    1.768    0.000 listlib.py:16(remove)
 22001998    1.140    0.000    1.140    0.000 {built-in method len}
```

233,525,220 function calls in 106.705 seconds

Ordered by: internal time
List reduced from 31 to 10 due to restriction <10>

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
194186759   58.607    0.000   58.607    0.000 profilednamaker.py:6(overlaps)
  4671030   39.505    0.000  103.272    0.000 profilednamaker.py:10(max_overlap)
  4671030    3.541    0.000    4.028    0.000 goody.py:17(irange)
     1814    1.650    0.001  104.921    0.058 profilednamaker.py:23(choose)
     3627    1.311    0.000    1.768    0.000 listlib.py:16(remove)
 22001998    1.140    0.000    1.140    0.000 {built-in method len}
  4671030    0.630    0.000    0.630    0.000 {built-in method min}
  1656204    0.155    0.000    0.155    0.000 profilednamaker.py:43(<lambda>)
  1658018    0.152    0.000    0.152    0.000 profilednamaker.py:42(<lambda>)
        1    0.010    0.010  106.699  106.699 profilednamaker.py:33(assemble)
```

Profiling Programs

- As we can see directly from the information above, the most `tottime` is spent in the overlaps function. It is very simple, so when I tried to write it another way, I couldn't get any time improvement. So, then I moved on to the `max_overlap` function. The `max_overlap` function calls overlaps for each possible overlap (based on lengths of the stands to match), so since the ratio of calls (`overlap calls/max_overlap` calls) is about 42/1, the average possible strand overlap is 42.

- By `numcalls` I realized from the place it was called (in choose) that I could simplify max_overlaps to not compute the maximum overlap, but just find (and immediately return) any overlap that exceeds a minimum specified in the choose method. By changing this, the above profile turned into the following one.

```
196057553 function calls in 87.748 seconds
```

```
Ordered by: internal time
  List reduced from 31 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
152048062   46.566    0.000   46.566    0.000 profilednamaker.py:6(overlaps)
  4671030   31.241    0.000   84.226    0.000 profilednamaker.py:10(exceeds_overlap)
  4671030    4.615    0.000    5.287    0.000 goody.py:17(irange)
     1814    1.713    0.001   85.939    0.047 profilednamaker.py:23(choose)
 26673028    1.334    0.000    1.334    0.000 {built-in method len}
     3627    1.316    0.000    1.795    0.000 listlib.py:16(remove)
  4671030    0.636    0.000    0.636    0.000 {built-in method min}
  1656204    0.158    0.000    0.158    0.000 profilednamaker.py:43(<lambda>)
  1658018    0.153    0.000    0.153    0.000 profilednamaker.py:42(<lambda>)
        1    0.009    0.009   87.743   87.743 profilednamaker.py:33(assemble)
```

Profiling Programs

- I was able to decrease the run time by about 20 seconds (almost 20%). Although `exceeds_overlap` (I changed the name from max_overlap) is called the same number of times as before, it calls overlaps about 25% fewer times, saving 12 seconds; and because it is called fewer times, `exceed_max` saves another 8 seconds over `max_overlap`, which together accounts for the full 20 seconds.

Profiling Programs

- In a large system with thousands of functions, it is a big win to use the profiler to focus our attention on the important functions: the ones that take a significant amount of time, and therefore the ones likely to cause major decrease in the runtime if improved. A rule of thumb is 20% of the code accounts for 80% of the execution time (some say 10%/90%, but the idea is the same). We need to be able to focus on which small amount of code the program spends most of its time in. In the code above, if by hard work I could make the bottom 28 functions run instantaneously and there would be at most a 6 second (7%) speedup: why bother? Better to have that code written as clearly as possible, since its execution accounts for so little time.

Profiling Programs

- Finally, the complete specifications for cProfile and pstats are available in the Python library documentation, under section 27. Debugging and Profiling, and under 27.4: The Python Profilers.

Hashing

LECTURE 4

Hashing: How Sets/Dicts are Faster than Lists for operations like in

- When we examined the complexity classes of various operations/methods on the `list`, `set`, and `dict` data-types/classes, we found that `sets/dicts` had many methods that were `O(1)`. We will briefly explain how this is accomplished, but a fuller discussion will have to wait until ICS-46. First, we will examine how hashing works and then analyze the complexity class of operations using hashing.
- Python defines a hash function that takes any object as an argument a produces a "small" integer (sometimes positive, sometimes negative) whose magnitude is 32 or 64 bits (depending on which Python you use). How this value is computed won't concern us now. But we are guaranteed (1) there is a fast way to compute it, (2) within the same program, an object with unchanged state always computes the same result for hashing, so immutable objects always compute the same result for their hash function.

Hashing

- Each class in Python implements a `__hash__` method, which is called by the hash function in Python's builtins module. Small integers are their own hash: hash(1) is 1; hash(1000000) is 1000000; but hash(1000000000000000000000000) is 1486940387. Even small strings have interesting hash values: hash('a') is 1186423063; hash('b') is 1561756564; hash('pattis') is -1650297348 (is your name hashed positive or negative)? (BUT SEE IMPORTANT NOTE BELOW).

Hashing

- Note that when we define our own classes, if we want them to be used in sets or as the keys in dictionaries (things that are hashed), we must define a `__hash__` method for that class (with only self as its parameter), and this method must return an int. If we do not provide a `__hash__` method and try to use objects from that class in sets or as the keys of dictionaries, Python will raise a **TypeError** with the message: "unhashable type".
- Technically, classes that contain mutator methods should NOT be hashable, but it is OK to define `__hash__` for a class of mutable objects: but if we do so, we must NEVER mutate an object while it is in a set or the key of dictionary: otherwise, the object might be "lost" because it is in the wrong bucket (see below). If we want to mutate such an object, we should remove it from the `set` or `dict`, mutate it, and then put it back in: all these operations are O(1).
- Again details about hashing is covered in much more detail in ICS-46.

Hashing

- Let's investigate how to use hashing and lists to implement `pset` (pseudo set) with a quick way to add/remove values, and check if a values is in the set: all are in complexity class `O(1)`.
- We define the class `pset` (and its `__init__`) as follows. Notice the `_bins` is a list of lists (starting with just 1 inner list, which is empty). The `__str__` method prints these bins.
- Objects in this class use `_len` to cache the number of values in their sets: incrementing it when adding values and decrementing when removing values. The last parameter specifies the load factor threshold, which we will discuss when we examine the add method. Notice that the first parameter, iterable, is anything we can iterate over, adding each value to the set to initialize it.

```
class pset:
    def __init__(self,iterable=[],load_factor_threshold=1):
        self._bins = [[]]
        self._len  = 0       # cache, so don't have to recompute from bins
        self._lft  = load_factor_threshold
        for v in iterable:
            self.add(v)

    def __str__(self):
        return str(self._bins)
```

Hashing

- Recall that `_bins` will store a list of lists (which we call a hash table). Each inner list is called a bin. Hash tables grow as we add values to them (just how they grow is the secret of the `O(1)` performance). Before discussing the add method, let's observe what _bins looks like as values are added.
- The load factor of a hash table is the number of values in the table divided by the number of bins (inner lists). As we add values to the bins, this number increases. Whenever this value exceeds the load factor threshold the number of bins is doubled, and all the values that were in the original bins are put back into the new bins (we will see why their positions often change). Such an operation is called rehashing. By increasing the number of bins, it will lower the load factor (by increasing its denominator) below the threshold.

```
0) start  : [[]]
1) add 'a': [['a']]
2) add 'b': [['b'], ['a']]
3) add 'c': [['b'], [], [], ['a', 'c']]
4) add 'd': [['b'], [], ['d'], ['a', 'c']]
5) add 'e': [[], [], [], ['c'], ['b'], ['e'], ['d'], ['a']]
6) add 'f': [['f'], [], [], ['c'], ['b'], ['e'], ['d'], ['a']]
7) add 'g': [['f'], [], [], ['c'], ['b'], ['e'], ['d'], ['a', 'g']]
8) add 'h': [['f'], [], ['h'], ['c'], ['b'], ['e'], ['d'], ['a', 'g']]
```

Hashing

Recall Load Factor = # values in the table/# of bins in the table

1) At the start, 1 bin with no values so the load factor is 0.
2) We add 'a' to the first bin in the _bins list; the load factor 1
3) We add 'b' to the first bin in the _bins list; the load factor is 2, which exceeds the threshold so all the values are rehashed as shown, and the load factor is now 2/2.
4) We add 'c' to a bin in the _bins list; the load factor is 3/2, which exceeds the threshold so all the values are rehashed (notice that 'a' and 'c' are both in the same bin; this is called a collision and often happens when there are many values in hash tables)
5) We add 'd' to the third bin in the _bins list; the load factor is 4/4.
6) We add 'e' to one bin in the _bins list; the load factor is 5/4, which exceeds the threshold so all the values are rehashed (notice all the   values are in their own bins now), and the load factor is now 5/8.
7) We add 'f' to the first bin in the _bins list; the load factor is 6/8.
8) We add 'g' to the last bin in the _bins list; the load factor is 7/8.
9) 8We add 'h' to the third bin in the _bins list; the load factor is 8/8. (adding another value will double the number of bins)

- Now let’s look at the `_bin` helper method, which finds the bin for the value: if the value is in the pset, it must be in that bin (although this calculation changes if the length of `_bins` changes, which is why rehashing is necessary); if the value is to be added to a `pset`, that is the bin it belongs in.

```
def _bin(self,v):
    return abs(hash(v))%len(self._bins)
```

- It hashes the value, computes its absolute value, and then computes its remainder when divided by the number of bins; so, the index it produces is always between `0` and `len(self._bins)-1`, is always a legal index for the _bins list. This is the bin the value belongs in WHEN THE HASH TABLE IS THAT LENGTH: if its length changes, the denominator in the calculation above also changes, so the bin it belongs in might change too.

Hashing

- The code for add is as follows:

```
def add(self,v):
    index = self._bin(v)
    if v in self._bins[index]:
        return
    self._len += 1
    self._bins[index].append(v)
    if self._len/len(self._bins) > self._lft:
        self._rehash()
```

Hashing

- It first computes the bin in which the value v must be in (if it is in the hash table) and then checks if it is there; if so, it returns immediately because sets have just one copy of a value. Otherwise, it increments `_len` and appends the value v into the bin in which it belongs. But if the newly added value makes the load factor exceed the threshold all the values are rehashed in the following helper method. The `_rehash` helper method is only called from add

```
def _rehash(self):
     old        = self._bins
     #double the number of bins (to drop the load_factor_threshold)
     #rehash old values: len(self._bins) has changed
     self._bins = [[] for i in range(2*len(old))]
     self._len = 0
     for bins in old:
         for v in bins:
             self.add(v)
```

Hashing

- This method remembers the old bins, resets the `_bins` and `_len`, and then adds each value v from the old bins into the new bins; its bin number might change because the % function calculated in `_bin`. By doubling the number of bins, there will be no calls to `_rehash` while all these values are added.
- Checking whether a value v is in a pset is simple: it just checks whether `v` is in the `bin/list` hashing says it belongs in.

```
def __contains__(self,v):
    return v in self._bins[self._bin(v)]
```

Hashing

- Likewise, removal goes to the bin the value v would be in IF it were in the `pset`, and if it is there it is removed and the cached length is decremented; if not in this bin, no others need to be checked no changes are made to the `pset`.

```
def remove (self,v):
    index = self._bin(v)
    for i in range(len(self._bins[index])):
        if self._bins[index][i] == v:
            del self._bins[index][i]
            self._len -= 1
            return
```

Hashing

- Finally, we can show the trivial `__len__` function, returning the cached values (incremented in add and decrmented in remove):

```
    def __len__(self):
        return self._len  # cached
```

- So, why are the add, contains, and remove methods O(1)?
- Because the hash function does a good job of randomizing in which bins values are stored, and the load factor is kept around 1 (meaning there are about as many bins as values: as more and more values are added, the length of the list of bins grows), the amount of time it takes to add/examine/remove a value from a its bin in hash table (as used in `pset`) is constant. That is, if there are N values in the `pset`, there are at lease N bins, and the average number of values in a bin is close to 1.

Hashing

- It takes a constant amount of work (independent from the number of values in the hash table) to hash a value to find its bin, and since each bin has about 1 value in it, it takes a constant amount of time to check or update that bin.
- Now, some bins are empty and some can have more than one value (but very few have a lot of values). I added an analyze method to `pset` so that it can show statistics about the number of bins and their lengths.
- If we call the following function as experiment(1000000), it generates 1 million random 26 letter strings and puts them in a hash table, whose size grows from 1 to 2**20 (which is a bit over a million)

Hashing

```
def build_set(n,m=26):
       s = pset()
       word = list('abcdefghijklmnopqrstuvwxyz'[:m])
       for i in range(n):
           random.shuffle(word)
           s.add(''.join(word))
       return s

   def experiment(n,m=26):
       s = build_set(n,m)
       s.analyze()
```

Hashing

- We can call this function to analyze the distribution of values in bins. Here is one result produced by calling experiment with the argument 1 million (whose output text is reduced a bit to fit nicely on one page).

```
bins with  0 values = 601,942 totaling       0 values; cumulative =         0
bins with  1 values = 148,927 totaling 148,927 values; cumulative =   148,927
bins with  2 values = 141,393 totaling 282,786 values; cumulative =   431,713
bins with  3 values =  90,011 totaling 270,033 values; cumulative =   701,746
bins with  4 values =  42,720 totaling 170,880 values; cumulative =   872,626
bins with  5 values =  16,507 totaling  82,535 values; cumulative =   955,161
bins with  6 values =   5,226 totaling  31,356 values; cumulative =   986,517
bins with  7 values =   1,411 totaling   9,877 values; cumulative =   996,394
bins with  8 values =     363 totaling   2,904 values; cumulative =   999,298
bins with  9 values =      59 totaling     531 values; cumulative =   999,829
bins with 10 values =      16 totaling     160 values; cumulative =   999,989
bins with 11 values =       1 totaling      11 values; cumulative = 1,000,000
```

Hashing

- As you can see, most bins store no values! But many other bins store just a few values; in fact, the bins storing 1-5 values account for over 95% of the values in the hash table. So, for almost 96% of the values in the hash table, it takes at most 5 comparisons to examine/update these bin, and 5 is a constant.
- Now we can close the circle started in this lecture by using Performance to empirically analyze whether all our conjectures about the performance of hash tables are correct. We will construct `pset`s with different numbers of values, doubling each time. We test each `pset` by adding N values and then performing N lookups. If each of these operations is truly `O(1)` and we do N of each, the complexity class of doing both it `O(N)`, so doubling N should double the time.

The data shows this behavior exactly, with much less error than our sorting analysis.

```
     N  |    Time   |  Ratio   |  Predicted    |  %Error
--------+-----------+----------+---------------+----------
  1,000 |  0.030    |          |   0.030             0
  2,000 |  0.060    |     2.0  |   0.060       |     0
  4,000 |  0.120    |     2.0  |   0.120       |     0
  8,000 |  0.240    |     2.0  |   0.241       |     0
 16,000 |  0.481    |     2.0  |   0.481       |     0  (predictions based on this run)
 32,000 |  0.962    |     2.0  |   0.962       |     0
 64,000 |  1.927    |     2.0  |   1.924       |     0
128,000 |  3.873    |     2.0  |   3.848       |     1
256,000 |  7.735    |     2.0  |   7.696       |     1
```

All this code is in the hashing.py module in the empirical project folder: a download that accompanies this lecture. So, you can run your own experiments.

Hashing

IMPORTANT:

- For some screwy reason whenever I am in a certain program, Python gives me exactly the same value when hashing a string; but when I stop the program and start a new one, it gives a different values (but always the same one for that run of the interpreter). That makes things much harder to explain. I think the hashing function uses some property of running the program (say the time it is started) in the hash function. This is good for exposing errors in code that uses hashing, but not so good for being able to show examples.
- This is also why different runs of exactly the same program with exactly the same data will produce different iteration orders for set.

Hashing

- Here are the methods that implement iteration for psets. The order that the values in the pset are produced is: all those value (in order) in the list in bin 0, all those values (in order) in the list in bin 1, etc.

```
def __iter__(self):
    for b in self._bins:
        for v in b:
            yield v
```

- Recall that the values moved around when rehashed. So that is why there is no simple order in the sets/dicts we iterate over.

Hashing

- Finally, we have discussed that set and dictionary keys cannot be mutable. Now we can get some insight why. If we put a value in its bin, but then change its state (mutate it), the `hash` function would compute a different result and `_bin` would probably want it in a different bin. And if it is in the wrong bin, looking for it, or trying to remove it, or trying to add it (with no duplicates) will not work correctly.

Sorting Data: actual data

LECTURE 5

Sorting 100,000 values

Analysis of 10 timings

```
avg = 0.041   min = 0.041  max = 0.042  span = 2.3%
```

Time Ranges

```
4.07e-02<>4.07e-02[ 40.0%]|**************************************************
4.07e-02<>4.08e-02[ 10.0%]|************
4.08e-02<>4.09e-02[ 10.0%]|************
4.09e-02<>4.10e-02[  0.0%]|A
4.10e-02<>4.11e-02[  0.0%]|
4.11e-02<>4.12e-02[ 10.0%]|************
4.12e-02<>4.13e-02[ 10.0%]|************
4.13e-02<>4.14e-02[  0.0%]|
4.14e-02<>4.15e-02[  0.0%]|
4.15e-02<>4.16e-02[ 10.0%]|************
4.16e-02<>4.17e-02[ 10.0%]|************
```

Sorting 200,000 values

Analysis of 10 timings

```
avg = 0.090   min = 0.089  max = 0.090  span = 2.1%
```

Time Ranges

```
8.86e-02<>8.88e-02[ 20.0%]|**************************************************
8.88e-02<>8.90e-02[ 10.0%]|*************************
8.90e-02<>8.92e-02[  0.0%]|
8.92e-02<>8.93e-02[ 10.0%]|*************************
8.93e-02<>8.95e-02[ 20.0%]|**************************************************A
8.95e-02<>8.97e-02[  0.0%]|
8.97e-02<>8.99e-02[ 10.0%]|*************************
8.99e-02<>9.01e-02[ 10.0%]|*************************
9.01e-02<>9.03e-02[  0.0%]|
9.03e-02<>9.05e-02[ 10.0%]|*************************
9.05e-02<>9.06e-02[ 10.0%]|*************************
```

Sorting 400,000 values

Analysis of 10 timings

```
avg = 0.194   min = 0.191  max = 0.197  span = 3.0%
```

Time Ranges

```
1.91e-01<>1.92e-01[ 20.0%]|**************************************************
1.92e-01<>1.93e-01[  0.0%]|
1.93e-01<>1.93e-01[ 10.0%]|*************************
1.93e-01<>1.94e-01[ 10.0%]|*************************
1.94e-01<>1.94e-01[ 20.0%]|**************************************************A
1.94e-01<>1.95e-01[ 10.0%]|*************************
1.95e-01<>1.96e-01[ 20.0%]|**************************************************
1.96e-01<>1.96e-01[  0.0%]|
1.96e-01<>1.97e-01[  0.0%]|
1.97e-01<>1.97e-01[  0.0%]|
1.97e-01<>1.98e-01[ 10.0%]|*************************
```

Sorting 800,000 values

Analysis of 10 timings

```
avg = 0.455   min = 0.440  max = 0.484  span = 9.7%
```

Time Ranges

```
4.40e-01<>4.45e-01[ 10.0%]|****************
4.45e-01<>4.49e-01[ 20.0%]|*********************************
4.49e-01<>4.54e-01[ 30.0%]|**************************************************
4.54e-01<>4.58e-01[ 10.0%]|****************A
4.58e-01<>4.62e-01[ 10.0%]|****************
4.62e-01<>4.67e-01[ 10.0%]|****************
4.67e-01<>4.71e-01[  0.0%]|
4.71e-01<>4.76e-01[  0.0%]|
4.76e-01<>4.80e-01[  0.0%]|
4.80e-01<>4.84e-01[  0.0%]|
4.84e-01<>4.89e-01[ 10.0%]|****************
```

Sorting 1,600,000 values

Analysis of 10 timings

```
avg = 1.013   min = 1.006  max = 1.025  span = 1.9%
```

Time Ranges

```
1.01e+00<>1.01e+00[ 20.0%]|*********************************
1.01e+00<>1.01e+00[ 10.0%]|****************
1.01e+00<>1.01e+00[  0.0%]|
1.01e+00<>1.01e+00[ 30.0%]|**************************************************A
1.01e+00<>1.02e+00[ 20.0%]|*********************************
1.02e+00<>1.02e+00[  0.0%]|
1.02e+00<>1.02e+00[ 10.0%]|****************
1.02e+00<>1.02e+00[  0.0%]|
1.02e+00<>1.02e+00[  0.0%]|
1.02e+00<>1.03e+00[  0.0%]|
1.03e+00<>1.03e+00[ 10.0%]|****************
```

Sorting 3,200,000 values

Analysis of 10 timings

```
avg = 2.296   min = 2.262  max = 2.360  span = 4.3%
```

Time Ranges

```
2.26e+00<>2.27e+00[ 10.0%]|****************
2.27e+00<>2.28e+00[ 30.0%]|**************************************************
2.28e+00<>2.29e+00[ 10.0%]|****************
2.29e+00<>2.30e+00[ 30.0%]|**************************************************A
2.30e+00<>2.31e+00[  0.0%]|
2.31e+00<>2.32e+00[  0.0%]|
2.32e+00<>2.33e+00[  0.0%]|
2.33e+00<>2.34e+00[ 10.0%]|****************
2.34e+00<>2.35e+00[  0.0%]|
2.35e+00<>2.36e+00[  0.0%]|
2.36e+00<>2.37e+00[ 10.0%]|****************
```

Sorting 6,400,000 values

Analysis of 10 timings

```
avg = 5.122   min = 5.093  max = 5.168  span = 1.5%
```

Time Ranges

```
5.09e+00<>5.10e+00[ 20.0%]|**************************************************
5.10e+00<>5.11e+00[ 10.0%]|*************************
5.11e+00<>5.12e+00[ 20.0%]|**************************************************
5.12e+00<>5.12e+00[ 10.0%]|*************************A
5.12e+00<>5.13e+00[ 10.0%]|*************************
5.13e+00<>5.14e+00[ 10.0%]|*************************
5.14e+00<>5.15e+00[  0.0%]|
5.15e+00<>5.15e+00[  0.0%]|
5.15e+00<>5.16e+00[ 10.0%]|*************************
5.16e+00<>5.17e+00[  0.0%]|
5.17e+00<>5.18e+00[ 10.0%]|*************************
```

Sorting 12,800,000 values

Analysis of 10 timings

```
avg = 11.622   min = 11.494  max = 11.786  span = 2.5%
```

Time Ranges

```
1.15e+01<>1.15e+01[ 20.0%]|**************************************************
1.15e+01<>1.16e+01[ 20.0%]|**************************************************
1.16e+01<>1.16e+01[  0.0%]|
1.16e+01<>1.16e+01[ 10.0%]|*************************
1.16e+01<>1.16e+01[ 10.0%]|*************************A
1.16e+01<>1.17e+01[  0.0%]|
1.17e+01<>1.17e+01[ 10.0%]|*************************
1.17e+01<>1.17e+01[ 10.0%]|*************************
1.17e+01<>1.18e+01[ 10.0%]|*************************
1.18e+01<>1.18e+01[  0.0%]|
1.18e+01<>1.18e+01[ 10.0%]|*************************
```

Sorting 25,600,000 values

Analysis of 10 timings

```
avg = 25.546   min = 25.288  max = 26.114  span = 3.2%
```

Time Ranges

```
2.53e+01<>2.54e+01[ 40.0%]|**************************************************
2.54e+01<>2.55e+01[ 10.0%]|************
2.55e+01<>2.55e+01[ 20.0%]|*************************
2.55e+01<>2.56e+01[  0.0%]|A
2.56e+01<>2.57e+01[ 10.0%]|************
2.57e+01<>2.58e+01[  0.0%]|
2.58e+01<>2.59e+01[ 10.0%]|************
2.59e+01<>2.59e+01[  0.0%]|
2.59e+01<>2.60e+01[  0.0%]|
2.60e+01<>2.61e+01[  0.0%]|
2.61e+01<>2.62e+01[ 10.0%]|************
```