Computer Science 1001.py

Lecture 7†: Integer greatest common divisor
Sequential vs. Binary Search
Defining Classes in Python

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University

Fall Semester, 2011/12
http://tau-cs1001-py.wikidot.com

Lecture 6: Highlights

- Randomized primality testing, expanded presentation.

- Cryptography: Secure communication over insecure communication lines.
- The discrete logarithm problem.
- One-way functions.
- Diffie-Hellman scheme for secret key exchange.

Lecture 6: Python Highlights

- The `random` package for generating (pseudo) randomness.
- The `random.randint(a,b)` function, producing a random integer in the closed interval `[a,b]` (both ends are included).

Lecture 7: Plan

- Integer greatest common divisor.
- Euclid's gcd algorithm.
- Using gcd statistics to approximate $6/\pi^2 \approx 0.6079271$.

- Searching in unordered lists and in ordered lists.
- Sequential search vs. binary search.

Integer Greatest Common Divisor

The greatest common divisor, or gcd, of two positive integers k, ℓ is the largest integer, g, that is a divisor of both of them. Since 1 always divides k, ℓ , the gcd g is well defined. If one of these two integers is zero, we define $\gcd(k, 0) = k$.

For example,

gcd(28,32)=4,

gcd(276,345)=69,

gcd(1001,973)=7,

gcd(1002,973)=1.

If $\gcd(k, \ell) = 1$, we say that k, ℓ are relatively prime.

If p is a prime number and $1 \leq k \leq p - 1$, then k, p are relatively prime. This observation is a key in establishing that each element in $\{1, 2, \ldots, p - 1\}$ has a multiplicative inverse $\mod p$ (this means that Z_p^* is a multiplicative group).

Computing Greatest Common Divisor Naively

The naive approach to computing $\gcd(k, \ell)$ is similar to the trial division approach: Start with $\min(k, \ell)$, and iterate, going down, testing at each iteration if the current value divides both k and ℓ.

How far do we go? Till the first divisor is found.

What is the (worst case) running time of this naive method? When relatively prime, the number of trial divisions is exactly $\min(k, \ell)$. If the minimum is an n bit number, the running time is $O(2^n)$. Hence this method is applicable only to relatively small inputs.

Slow GCD Code

```
def slow_gcd(x,y):
    """ greatest common divisor of two integers -
    naive inefficient method """
    assert isinstance(x,int) and isinstance(y,int)
     # type checking: x and y both integers
    x,y=abs(x),abs(y) # simultaneous assignment to x and y
            # gcd invariant to abs. Both x,y now non-negative
    if x<y:
        x,y=y,x    # switch x and y if x < y. Now y <= x
    for g in range(y, 0, -1): # from x downward to 1
        if x%g == y%g == 0:  # does g divide both x and y?
            return g
    return None # should never get here, 1 divides all

def slow():
    return slow_gcd(10**7+128,10**6-96)

>>> import timeit
>>> slow_gcd(10**7+128,10**6-96)
32
>>> timeit.timeit(slow,number=1)  # just one run
0.5497519969940186       # just above half a second
```

Computing GCD – Euclid's Algorithm

Euclid, maybe the greatest known early Greek mathematician, lived in Alexandria in the 3rd century BC. His book, *Elements*, lays the foundation to so called Euclidean geometry, including an axiomatic treatment. The book also deals with number theory and describes an efficient gcd algorithm.

(drawing from Wikipedia)

Euclid's gcd algorithm is based on the following invariant (an invariant is a property that remains true, or a value that is unchanged, before and after applying some transformation): Suppose $0 < \ell < k$, then $\gcd(k, \ell) = \gcd(k \mod \ell, \ell)$ (a formal proof is deferred).

The algorithm replaces the pair (k, ℓ) by $\gcd(k \mod \ell, \ell)$, and keeps iterating till the smaller of the two reaches zero. Then it uses the identity $\gcd(h, 0) = h$.

Computing GCD – Euclid's Algorithm (cont.)

Euclid's gcd algorithm is based on the following invariant:
Suppose $0 < \ell < k$, then $\gcd(k, \ell) = \gcd(k \mod \ell, \ell)$.

Notice that after taking the remainder, $k \mod \ell$ is strictly smaller than ℓ. Thus one iteration of this operation reduces both numbers to be no larger than the original minimum.

It can be shown that two iterations of this operation make the numbers smaller than half the original maximum.

Example:
k_0=4807526976, ℓ_0=2971215073
k_1=2971215073, ℓ_1=1836311903
k_2=1836311903, ℓ_2=1134903170
k_3=1134903170, ℓ_3=701408733
k_4=701408733, ℓ_4=433494437

Suppose that originally k is an n bit number, namely $2^{n-1} \leq k < 2^n$. On every second iteration, the maximum number is halved. So in terms of bits, the length of the maximum becomes at least one bit shorter. Therefore, the number of iterations is at most $2n$.

Python Code – Euclid's Algorithm, Displaying

The following code computes `gcd(x,y)` using Euclid's algorithm. In addition. it prints all intermediate pairs.

```
def display_gcd(x,y):
    """ greatest common divisor of two integers, Euclid's algorithm.
    This function prints all intermediate results along the way. """
     assert isinstance(x,int) and isinstance(y,int)
            # type checking: x and y both integers
     x,y=abs(x),abs(y) # simultaneous assignment to x and y
              # gcd invariant to abs. Both x,y now non-negative
     if x<y:
         x,y=y,x    # switch x and y if x < y. Now y <= x
     print(x,y)
     while y>0:
         x,y=y,x%y
         print(x,y)
     return x
```

Python Code – Euclid's Algorithm, Displaying (run)

```
>>> display_gcd(10946,6765)
10946 6765
6765 4181
4181 2584
2584 1597
1597 987
987 610
610 377
377 233
233 144
144 89
89 55
55 34
34 21
21 13
13 8
8 5
5 3
3 2
2 1
1 0
1     # final outcome -- gcd(10946,6765)
```

Non trivial question: Which pairs of n bit integers, `x,y`, cause a worst case performance (maximal number of iterations) for Euclid's gcd?

Python Code – Euclid's vs. Slow

As noted before, on n bit numbers, Euclid's algorithm takes at most $2n$ iterations, while slow gcd takes up to 2^n iterations.

Let us put this theoretical analysis to the ultimate test – the test of the clock. We note that we now consider a version of Euclid's algorithm which does not display intermediate results (code omitted).

```
>>> slow_gcd(10**7+128,10**6-96)
32
>>> len(bin(10**6-96))
22 # 10**6-96 is 22 bits long

>>> def slow():
return slow_gcd(10**7+128,10**6-96)

>>> def euclid():
return gcd(10**7+128,10**6-96)

>>> timeit.timeit(slow,number=1)  # one run
0.5098249912261963  # half a second

>>> timeit.timeit(euclid,number=100000) # 100,000 runs
0.18328499794006348 # one fifth of a second
```

So, in this case at least, theory and practice do agree.

Extended GCD

Claim: if $\gcd(x, y) = g$, then there are two integers a, b such that $g = ax + by$. For example,
$\gcd(1001, 973) = 7$, and indeed $35 \cdot 1001 - 36 \cdot 973 = 7$,
$\gcd(100567, 97328) = 79$. Indeed $601 \cdot 100567 - 621 \cdot 97328 = 79$.
$\gcd(10^7, 10^6 + 1) = 1$, and indeed $10^5 \cdot 10^7 - 999999 \cdot (10^6 + 1) = 1$.

A simple modification of Euclid's gcd algorithm enables to compute these coeeficients a, b efficiently. This algorithm is termed extended Euclidian gcd.

If p is a prime and $1 \leq x \leq p - 1$ (this is also denoted $x \in Z_p^*$), then $\gcd(x, p) = 1$ (why?).

Therefore there are integers a, b such that $ax + bp = 1$. In particular, we have $ax = 1 \mod p$. Therefore a is the multiplicative inverse of x modulo p. (This establishes that Z_p^* is a group).

Mathematical Gedankenexperiment

Choose two positive integers, x, y, independently at random. What is the probability that they are relatively prime, *i.e.* $\gcd(x, y) = 1$?
We note that uniform choice over the positive integers is not possible (think why!). But we could choose x, y uniformly over a large range [1,N], and then send $N \longrightarrow \infty$.

Let us perform a small Gedankenexperiment. The probability that 2 divides x is $1/2$, and so is the probability that 2 divides y. By independence, 2 divides both x and y with probability $1/4$.
Likewise, the probability that 3 divides both x and y is $1/9$, the probability that 5 divides both x and y is $1/25$, etc.

Mathematical Gedankenexperiment, cont.

Define the event $E_p = \{(x, y) \mid$ the prime p divides both x and $y\}$.

We argued that $Pr(E_p) = 1/p^2$, so $Pr(\overline{E_p}) = 1 - 1/p^2$.

$\gcd(x, y){=}1$ if and only if $(x, y) \in \overline{E_2} \cap \overline{E_3} \cap \overline{E_5} \cap \ldots = \bigcap_{p \text{ prime}} \overline{E_p}$.

With some leap of faith, all these events are mutually independent.

$$
\begin{aligned}
Pr(\gcd(x, y){=}1) &= \prod_{p \text{ prime}} \left(1 - 1/p^2\right) \\
&= \left(\sum_{n=1}^{\infty} 1 + 1/n^2\right)^{-1} \\
&= 6/\pi^2 \approx 0.6079271
\end{aligned}
$$

(the last two equalities are from standard calculus class).

Experimental Math: Code

We can approximate $Pr(\gcd(x,y)=1)$ by generating many random pairs (x,y) (over a large range), and count how many of them are relatively prime.

```
def samplegcd(n,iternum):
    """ repeats the following iternum times: Choose two
    random integers, n bits long each (leading zeroes OK).
    Check if they are relatively prime. Compute the frequency"""

    count=0
    for i in range(0,iternum):
        if gcd(random.randint(1,2**n-1),random.randint(1,2**n-1))==1:
            count += 1
    return(count/iternum)
```

Experimental Math: Execution

We can approximate $Pr(\gcd(x,y)=1)$ by generating many random pairs (x,y) (over a large range), and count how many of them are relatively prime.

```
>>> import random
>>> for i in range(1,9):
print(samplegcd(40,2**(20+i)))

0.608121395111
0.608100652695
0.608383536339
0.607866346836
0.607976645231
0.608000293374
0.608054250479
0.607975456864
```

Comments:

- Returned values converge to the "true value", 0.6079271, but not monotonically.
- These executions take very long (a couple of hours for $2^{28} = 268,435,456$ samples).

Search

(taken from http://bizlinksinternational.com/web/web%20seo.php)

Search

Search has always been a central computational task. In early days, search supposedly took one quarter of all computing time.

The emergence and the popularization of the world wide web has literally created a universe of data, and with it the need to pinpoint information in this universe.

Various search engines have emerged, to cope with this challenge. They constantly collect data on the web, organize it, and store it in sophisticated data structures that support efficient (fast) access, resilience to failures, frequent updates, including deletions, etc., etc.

In this class we will deal with two much simpler data structures that support search:

- unordered list
- ordered list

Representing Items in a List

We assume our data is arranged in a list. Recall that in Python, a list with n elements is simply a mapping from the set of indices, $\{0, \ldots, n-1\}$, to a set of items.

In our context, we assume that items are records having a fixed number of information fields. For example, our `Student` items will include the two fields each: `name, identity number`.

We can arrange each item as a tuple with two entries (corresponding to the example above). But this is not what we are going to do.

Representing Students' Records in a List

The following list was generated manually from the ~~106~~ 109 strong list of students in this class. To protect your privacy, only first names are given (hopefully spelled correctly). The id numbers in these records were generated at random.

(Bear this in mind when you apply to get your new biometric ID card.)

```
>>> len(students_list)
12
>>> for i in range(12):
print(students_list[i])

[Yuval,23094355]
[Shady,55568503]
[Lielle,37738993]
[Tal,38136128]
[Rina,24004660]
[Barak,42983984]
[Gal,50804902]
[Noa,58329396]
[Roee,28016070]
[Amir,51271851]
[Yana,22547403]
[Or,41155245]
```

The Student Class

In fact, every entry in the list is of type Student, a class defined by us. Classes are heavily used in the object oriented programming, including OOP facets of Python. At this point we will just give some details, while more elaborate discussion will follow later.

```
>>> type(students_list[0])
<class '__main__.Student'>
```

Objects of the class have two fields: name and id.

```
>>> students_list[0].name
'Yuval'
>>> students_list[0].id
23094355
>>> students_list[7].name
'Noa'
>>> students_list[7].id
58329396
```

Searching the List

We are now interested in searching the list. For example, we want to know if a student called Yuval is in the list, and if so, what is his/her ID number is. In this example, the student's name we look for is the key, and the associated ID is the value we are interested in.

With such an unordered list, we have no choice but to search for items sequentially, one by one, in some order. For example, by going over the list from the first entry, `students_list[0]`, to the last entry, `students_list[11]`.

What is the best case running time of sequential search? Worst case running time?

Food for thought: Would it be better to sample items at random? (think of best, worst, and average cases).

Sequential Searching: Code

```
def sequential_search(key,lst):
    """ sequential search from lst[0] till last lst element
        lst need not be sorted for sequential search to work """
    for elem in lst:
        if elem.name==key:
            return elem
    else:    #  we get here when the key is not in the list
        print(key, "not found")
        return None
```

Searching backwards

```
def sequential_search_back(key,lst):
    """ sequential search from last lst element till first one.
        lst need not be sorted for sequential search to work """
    l=len(lst)
    for i in range(l-1,-1,-1):
        if lst[i].name==key:
            return lst[i]
    else:    #  we get here when the key is not in the list
        print(key, "not found")
        return None
```

Why not just `sequential_search(lst[::-1],value)`?
(think what will happen to best case inputs.)

Running the Code

```
>>> sequential_search("Or",students_list)
[Or,41155245]
>>> sequential_search("Benny",students_list)
Benny not found       # Benny is not a student anymore
>>> sequential_search("Shady",students_list)
[Shady,55568503]

>>> sequential_search_back("Or",students_list)
[Or,41155245]
>>> sequential_search_back("Benny",students_list)
Benny not found    # not a student even when searching backwards
>>> sequential_search_back("Shady",students_list)
[Shady,55568503]
```

What keys cause worst case running time for both forward and backward sequential searches?

Sequential Search: Time Analysis

Any sequential search in an unordered list goes over it, item by item. If the list is of length n, sequential search will take n steps in the worst case (when the item is not found because it is missing).

For our exclusive (thus short) list of students, this is not a problem. But if n is very large, such a search will take very long.

Search in Unordered vs. Ordered Lists

Hands on experience: Searching for a word in a book vs. searching for it in a dictionary.

(We mean a real world, hard copy, dictionary, not Python's `dict`!)

Sequential vs. Binary Search

For unordered lists of length n, in the worst case, a search operation compares the key to all list items, namely n comparisons.

On the other hand, if the n element list is sorted, search can be performed much faster. We first compare input key to the key of the list's middle element, an element whose index is $\lfloor n/2 \rfloor$.

- If the input key equals the middle element's key, we return the middle element and terminate.
- If the input key is greater than the middle element's key, we can restrict our search to the top half of the list (indices from $\lfloor n/2 \rfloor + 1$ to $n - 1$).
- If the input key is smaller than the middle element's key, we can restrict our search to the bottom half of the list (indices from 0 to $\lfloor n/2 \rfloor - 1$).

Time Analysis of Binary Search

At each stage, we either terminate or cut the size of the remaining list by half. This is why this process is termed binary search.

- We start with an ordered list of length n.
- If $n = 1$, we compare the only item in the list to the key, and terminate.
- If $n > 1$, we either terminate in one step (if the key is found), or continue to look for the key in a list of length $\lfloor (n-1)/2 \rfloor$.
- In each iteration we perform one comparison, and cut the length by one half, till the length reaches $n = 1$.
- The number of times we can halve n till we reach 1 is $\lceil \log_2(n) \rceil \leq 1 + \log_2(n)$.
- So the (worst case) time complexity of binary search is $1 + \log_2(n)$. For large n, it is much faster than sequential search.
- Of course, binary search requires preprocessing (sorting the list).

Binary Search: Python Code

```
def binary_search(key,lst):
    """ iterative binary search
        lst better be sorted for binary search to work """
    n=len(lst)
    lower=0
    upper=n-1
    while lower<upper:
        middle=(upper+lower)//2
        if key==lst[middle].name:     # item found
            return lst[middle]
        elif key<lst[middle].name:    # item cannot be in top half
            upper=middle-1
        else:                 # item cannot be in bottom half
            lower=middle+1
    if lower==upper:      # list reduced to size 1
        if lst[lower].name==key:
            return lst[lower]
        else:    #  we get here when the key is not in the list
            print(key, "not found")
            return None
```

Binary Search: Preprocessing

As a sanity check, we first run the code on the small `students_list` of length 12, used above to test the sequential search code.

```
>>> students_list
[[Yuval,23094355], [Shady,55568503], [Lielle,37738993], s
[Tal,38136128], [Rina,24004660], [Barak,42983984],
[Gal,50804902], [Noa,58329396], [Roee,28016070],
[Amir,51271851], [Yana,22547403], [Or,41155245]]
```

To apply binary search, we better sort the list first. We want to sort the items by their names. We employ a built-in sorting function, `sorted`, and tell it to use the name as the key to be sorted (this part will be clarified when we define the `greater than` relation with respect to the Student Class, in the next lecture).

```
>>> sorted_list=sorted(students_list)

>>> sorted_list
[[Amir,51271851], [Barak,42983984], [Gal,50804902],
[Lielle,37738993], [Noa,58329396], [Or,41155245],
[Rina,24004660], [Roee,28016070], [Shady,55568503],
[Tal,38136128], [Yana,22547403], [Yuval,23094355]]
```

Binary Search: Running the Code

```
>>> binary_search("Or",sorted_list)
[Or,41155245]
>>> binary_search("Benny",sorted_list)
Benny not found
>>> binary_search("Shady",sorted_list)
[Shady,55568503]
```

What happens if we try running binary search on an unsorted list?

```
>>> binary_search("Or",students_list)
Or not found
```

This should not come as a surprise.

The running time of sequential and binary search for short lists (*e.g.* of length 12, namely $1 + \log_2(12) = 4$ vs. 12) are not easy to tell apart.

The difference is distinguishable when considering longer lists, *e.g.* of length $n = 10^7 = 10,000,000$.

Sequential Search: Timing the Code on Long Lists

We could have based our list on the leaked Israeli population registry. However, to avoid potential legal troubles, the names and identity numbers were generated completely at random (details – later).

```
>>> large_stud_list=students(10**7)
          # generates a list with random names and id numbers
>>> large_stud_list[7*10**6]
[Jqcirv Iymejdh,24298269]
>>> sequential_search("Jqcirv Iymejdh",large_stud_list)
[Jqcirv Iymejdh,24298269]
>>> sequential_search("Benny",large_stud_list)
Benny not found

>>> def seq():
return sequential_search("Benny",large_stud_list)

>>> import timeit
>>> timeit.timeit(seq,number=100)  # 100 runs
144.78874683380127
```

So, one hundred sequential searches in a $10,000,000$ long list for a non-existing key took about 145 seconds.

Note: Printing `"Benny not found"` was disabled (why?).

Binary Search: Timing the Code on Long Lists

We sort the list, and time the search for the same non-existing key

```
>>> large_sorted_list=sorted(large_stud_list,
        key = lambda student: student.name)
>>> large_sorted_list[7*10**6]
[Sex Prbwe,29327748]    # note: I did not make this up!
>>> large_stud_list[7*10**6]
[Jqcirv Iymejdh,24298269]
>>> binary_search("Jqcirv Iymejdh",large_sorted_list)
[Jqcirv Iymejdh,24298269]
>>> binary_search("Benny",large_sorted_list)
Benny not found

>>> def binary():
binary_search("Benny",large_sorted_list)
>>> timeit.timeit(binary,number=100)  # 100 runs
0.0025060176849365234
```

So, one hundred binary searches in a $10,000,000$ long list for a non-existing item took about 0.0025 seconds. This is 57,776 times faster than sequential search.
Theoretically, one would expect a $10^7/\log_2(10^7) \approx 384,615$ ratio.

Binary Search: A High Level View

Binary search is widely applicable (not only for searching a list). In general, when we look for an item in a huge space, and that space is structured so we could tell if the item is

1. right at the middle,
2. in the top half of the space,
3. or in the lower half of the space.

In case (1), we solve the search problem in the current step. In cases (2) and (3), we deal with a search problem in a space of half the size.

In general, this process will thus converge in a number of steps which is $\log_2$ of the size of the initial search space. This makes a huge difference. Compare the performance of binary search to that of going sequentially over the original space, item by item.

Binary Search and the Feldenkrais Method

In general, binary search terminates (with either an item matching the key, or the response that none exists) in a number of steps which is $\log_2$ of the size of the initial search space. This makes a huge difference. Compare the performance of binary search to that of going sequentially over the original space, item by item.

Borrowing from the late Dr. Moshé Feldenkrais, it "...makes the impossible possible, the possible comfortable, and the comfortable pleasant."

We will encounter binary search again soon, when dealing with finding roots of real functions, confronting nightmares, and more.

Last, But Not Least

We have already imported and used different Python's packages, such as `random`, `timeit`, and `math`.

Of equal importance is Python's `antigravity` package, used for levitating.

To employ it, simply type `import antigravity` in your IDLE shell screen.