Homework A

Homework A for avalg11
Peter Boström <pbos@kth.se>
2011-10-04

Problem 1: Show that the following algorithm will compute GCD of two non-negative integers a and b where one of them is at least 1.

```
gcd(a,b):
    # gcd(a,b) = gcd(b,a), this makes sure a >= b during the rest of the algorithm
    if b > a
        return gcd(b, a)

    # base case, gcd(a, 0) = a
    else if b == 0
        return a

    # if 2 is a factor in both, 2 is obviously a factor in the gcd, remove 2 from both a and b.
    else if a and b are even
        return 2*gcd(a/2, b/2)

    # any excess factors of 2 available in a or b (not both) can be divided away as they won't be part of the gcd.
    else if a is even
        return gcd(a/2, b)
    else if b is even
        return gcd(a, b/2)

    # d | a, d | b => d | a-b, d | b
    # gcd of a and b must be the gcd of b and a-b as well
    else
        return gcd(b, a-b)
```

d | a, d | b means a = l*d, b = k*d where k and l are both integers
a-b = l*d - k*d = (l-k) * d, as l and k are both integers, a-b = m*d, where m is also an integer
Thus: d | a, d | b => d | b, d | a-b

gcd(a,b) | a, gcd(a,b) | b => gcd(a,b) | b, gcd(a,b) | a-b
Hence: gcd(a,b) = gcd(b, a-b)

- Time/bit complexity

gcd(a,b) = gcd(b,a) reduces no bits, but will never occur more than once in a row

gcd(a,0) = a, base case (return a)

2*gcd(a/2, b/2) reduces a and b with one bit each (that is, total number of bits with 2)

gcd(a/2, b) reduces the total number of bits by one
gcd(a, b/2) reduces the total number of bits by one

gcd(a,b) = gcd(b, a-b) will only occur when a,b are odd.
    => a-b even
    => might turn gcd(b, a-b) to gcd(a-b, b) i the following step
    => in the following step a-b will get divided by 2, reduces the number of bits by one.

In the worst case 4 iterations are required before one bit is removed, 4 iterationer innan en bit försvinner. (gcd(b, a) => gcd(a, b) => gcd(b, a-b) => gcd(a-b, b) => gcd((a-b)/2, b))

The time complexity (with unit cost) is therefore 4*O(n) = O(n), where n is the number of bits in a and b together.

- Time complexity without unit cost

O(n) is without unit cost. Division with 2 is a bitshift by one, which can be implemented in O(n), as can a-b. All iterations occur in O(n) which yields O(n)*O(n) => $O(n^2)$ in total.

Problem 2: Chinese remainder theorem

```
    7693294541716125369

    n = 8902240814

    Chinese remainder theorem gives

    x = a*n*r1 + b*(n+1)*r2

    where a and b are obtained from the extended euclidian algorithm with
    an + b(n+1) = gcd(n, n+1) = 1
```

== problem2.py ==

```
#!/usr/bin/env python3
def egcd(a, b):
    u, u1 = 1, 0
    v, v1 = 0, 1
    g, g1 = a, b
    while g1:
        q = g // g1
        u, u1 = u1, u - q * u1
        v, v1 = v1, v - q * v1
        g, g1 = g1, g - q * g1
    return u, v, g

n = 8902240814
print('n =', n)
(a, b, gcd) = egcd(n, n+1)

assert gcd == 1

# ua + vb = gcd(a, b) = 1
# ua = 1 - vb mod(b) = ua = 1 mod(b)
# vb = 1 - ua mod(a) = vb = 1 mod(a)

# ua*r2 + vb*r1 mod(b) <=> r2 mod(b)
# ua*r2 + vb*r1 mod(a) <=> r1 mod(a)

r1 = 123456789
r2 = 987654321
x = a*n*r1 + b*(n+1)*r2

print(x)
print(x % n)
print(x % (n+1))
```

Problem 3: Design an algorithm that sorts n non-negative integers in linear time on a unit cost RAM if all elements are of magnintude at most n^{10}.

LSD radix sort with base n, of non-negative integers magnitude less than n^m at most, gives O(n*m) complexity.

If m is fixed to 10, that gives a time complexity of O(n*10), or O(n). In either case where m is fixed, the algorithm is O(n). If n^{10} should be included, we can set m = 11, which still gives O(n).

== radix.py ==

```
#!/usr/bin/env python3
from collections import deque

def radix(num): # Sorts n non-negative integers of magnitude n^m in O(n*m)
    n = len(num)

    buckets = []
    for i in range(n):
```

```
            buckets.append(deque())

        nums = deque()

        for x in num:
            nums.append((x,x))

        # This will run for m times if all numbers are < n^m
        while True:

            # O(n) (n numbers, append/popleft constant with linked list)
            # mod and integer divisions are unit cost (O(1)).
            while nums:
                (i, rem) = nums.popleft()
                idx = rem % n;
                rem //= n;
                buckets[idx].append((i, rem))

            non_nulls = 0
            # For each bucket (O(n)) + Put each number back in combined list (O(n))
            for i in range(n):
                if len(buckets[i]) > 0:
                    non_nulls += 1
                    nums.extend(buckets[i])
                    buckets[i].clear()

            # If all numbers were in the same bucket, sorting is done..
            if non_nulls == 1:
                break

        # put all numbers back in the original list (O(n))
        idx = 0
        for i in range(n):
            (foo, bar) = nums.popleft()
            print(foo)
            num[i] = foo

        return num


    # Problem 4: Design an algorithm that sorts n comparable elements in time O(n log m) provided we know that there are only m distinct values.

    This algorithm operates likes counting sort, except counting isn't done with a table. It needs to count m distinct values in O(n log m) time.

    Using a self-balancing tree with guaranteed O(n log n) for search and insertion, let's make this as a map (a "balanced-tree map"), like hash table => hash map.

    input: [x] (a list of n integers with m distinct values)

    # This tree will never contain more than m distinct values.
    tree = balanced_tree_map() # (number used as key, count as value)

    for i in x: # Run n times, O(n*2*log m) = O(n log m) in total
        if i not in tree: # O(log m)
            tree[i] = 1 # insert number with count (O(log m)
        else: # increase count of how many times i has been seen
            tree[i] += 1 # O(log m)

    # put them back (exactly like like counting sort)
    idx = 0
    for (key, count) in tree: # in-order traversal of tree map
        for i in range(count): # amortized this will in total be run n times
            x[idx] = key # And each iteration is constant, O(1)
            idx += 1
```

```
return x # optional, sorting has been done in-place.
```

Problem 5: Show that resolution yields a polynomial time algorithm that determines the (un)satisfiability of 2-CNF formulas by deriving new clauses until either an empty clause is obtained or no more clauses can be derived.

2CNF clauses can only derive other 2CNF clauses. m symbols can at most be paired up with m other symbols, at most $4*m^2-4m+2m$, or in either case $O(m^2)$ possible 2CNF or 1CNF clauses in total. From n clauses at most 2n symbols can exist, which gives a max of $O(n^2)$ possible clauses as well.

Deriving an empty clause means the formula is unsatisifiable. If no more clauses can be derived, it's satisfiable.

```
for each clause c1: # O(n²)
    for each clause c2 != c1: # O(n²)
        if c1 and c2 can be used to derive a new clause: # O(1)
            derive new clause c3 # O(1)
            if c3 is empty: # O(1)
                return UNSATISFIABLE # O(1)
            if c3 doesn't exist already: # O(n²)
                add c3 to formula # O(1)

# no more clauses can be derived
return SATISFIABLE
```

Time taken in total: $O(n^2)*O(n^2)*O(n^2) = O(n^6) \in P$

(This can definitely be improved upon.)