CS 217 – Algorithm Design and Analysis

Shanghai Jiaotong University, Spring 2020

Yujie Lu Jiabao Ji Tong Chen

Handed out on 2020-03-05
First submission and questions due on 2020-03-10
You will receive feedback from the TA.
Final submission due on 2020-03-17 before class

1 Bit Complexity, Recursion, and Dynamic Programming

1.1 Bit Complexity of Euclid's Algorithm

We have proved that Euclid's algorithm for computing $\gcd(a, b)$ makes at most $O(\log a)$ iterations. What is the overall running time? Each iteration computes $u \mod v$ for some integers. This can be done by integer division. What is its running time? There are very sophisticated algorithms, but python probably does not come with them. Recall the "school method" for dividing integers. Have a look at the pdf slides on the webpage for an illustration of the school method. It is especially simple if we are dealing with binary numbers. If a and b have at most n bits, then the school method has complexity $O(n^2)$.

Exercise 1 Show the following, more precise bound of the school method for integer division: If a has n bits and b has k bits, then the school method can be implemented to run in $O(k(n-k))$ operations.

Proof: First, we know $n > k$ since $a > b$. To get r, $r = a\%b, (0 \leq r < b)$, so the bits of r, we write it as b_r. $b_r < k$.

Remind the procedure of brute-force algorithm of $a\%b$, in each iteration, n decreases(since we can drop the higher bits). In the final iteration, $n < k$, also this n is exactly b_r, so we don't need to do the iteration n times, but only $n - k$ times. And each iteration costs n subtraction operation, which is an integer operation, in all we do no more than $n(n - k)$ times of basic operations. $\square$

Exercise 2 Show that the bit complexity of Euclid's algorithm, using the school method to compute $a \mod b$, is $O(n^2)$. That is, if a and b have at most n bits, then $\gcd(a, b)$ makes $O(n^2)$ bit operations.

In order to do so, here is python code of the Euclidean algorithm:

```
def euclid(a,b):
  while (b > 0):
      r = a % b # so a = bu+r
      if (r == 0):
          return b
      s = b % r # so b = rv + s
      a = r
      b = s
  return a
```

Don't be afraid to introduce notation! I recommend to let n denote the number of bits of a. Take some other letters for the number of bits in b and so on.

Proof: Assume that $\gcd(a, b)$ takes t iterations and $t \leq 2n$. To better represent our ideas. Let's introduce some notations. Conside a process of iterations as follows:

$$\gcd(a, b) = \gcd(x_t, x_{t-1}) \rightarrow \gcd(x_{t-1}, x_{t-2}) \rightarrow \cdots \rightarrow \gcd(x_1, 0) .$$

And $\forall 1 \leq k \leq t$ let x_k has y_k bits. We know $y_t \leq n$ plus $x_k > x_{k-1}$. Hence $\forall k, y_k \geq y_{k-1}$.

Let the operations of all iterations be T. And the operations of kth iteration be T_k. Now apply the conclusion of Exercise 1, we have

$$\begin{aligned}
T &= \sum_{k=1}^{t} T_k \\
&\leq \lambda \sum_{k=1}^{t} y_k \left(y_k - y_{k-1}\right) \\
&\leq \frac{\lambda}{2}\left(\sum_{k=1}^{t} \left(y_k - y_{k-1}\right)^2 + y_t^2\right) \\
&\leq \frac{\lambda}{2}\left(\sum_{k=1}^{t} \left(y_k - y_{k-1}\right)\right)^2 + \frac{\lambda}{2} y_t^2 \\
&\leq \frac{\lambda}{2} n^2.
\end{aligned}$$

Hence gcd (a, b) makes $O\left(n^2\right)$ operations. □

1.2 Computing the Binomial Coefficient

Next, we will investigate the binomial coefficient $\binom{n}{k}$, which you might also know by the notation C_n^k. The number $\binom{n}{k}$ is defined as the number of subsets of $\{1, \ldots, n\}$ which have size exactly k. This immediately shows that $\binom{n}{k}$ is 0 if k is negative or larger than n. You might have seen the following recurrence:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ if } n, k \geq 0 \ .$$

Exercise 3 [A Recursive Algorithm for the Binomial Coefficient] Using pseudocode, write a recursive algorithm computing $\binom{n}{k}$. Implement it in python! What is the running time of your algorithm, in terms of n and k? Would you say it is an efficient algorithm? Why or why not?

Proof: recursive code:

```
def recursiveCompute(n, k):
    if (n <= k or k == 0):
        return 1
    else:
        return recursiveCompute(n - 1, k - 1) + recursiveCompute
                                                    (n - 1, k)
```

Like recursive algorithm for computing Fib(n), we can get a recursive tree, which has $\binom{n}{k}$ leaves and $\left(2\binom{n}{k}-1\right)$ nodes in all. We can prove it by induction.

Firstly, prove it for a fixed n, to do that, we use an induction. And suppose the equation is true for all $m < n$ (base $n = 1$ is trivial).

Base: $k = 0$:

Obviously, the recursive tree only has 1 node, which satisfies the equation.

Induction:

To compute $\binom{n}{k+1}$, we will add a node for $\binom{n}{k+1}$ as the new root, and two sub recursive tree for computing $\binom{n-1}{k}$, $\binom{n-1}{k+1}$, whose nodes size is known by induction hypothesis. The new tree has

$$\binom{n-1}{k} + \binom{n-1}{k+1} = \binom{n}{k+1}.$$

leaves, and

$$\left(2\binom{n-1}{k} - 1\right) + \left(2\binom{n-1}{k+1} - 1\right) + 1 = 2\binom{n}{k+1} - 1.$$

nodes.

Similar to the induction above, we can tell for any n, k, the result holds. For the running time analysis, each node in the tree needs to do an addition, so it's $O\left(2\binom{n}{k} - 1\right) = O\left(\binom{n}{k}\right)$.

It's not a good algorithm, since the time complexity is very high, and it does a lot of redundant addition. □

Exercise 4 [A Dynamic Programming Algorithm for the Binomial Coefficient] Using pseudocode, write a dynamic programming algorithm computing $\binom{n}{k}$. Implement it in python! What is it running time in terms of n and k? Would you say your algorithm is efficient? Why or why not?

Proof: code:

```
def dpCompute(n, k):
    a = [[0 for x in range(k + 1)] for y in range(n + 1)]
    for i in range (n + 1):
        a[i][0] = 1
        for j in range (1, min(i, k) + 1):
            a[i][j] = a[i - 1][j - 1] + a[i - 1][j]
    return a[n][k]
```

The algorithm needs to do n iterations, in each iteration, it needs to do $\min(i,k)$ additions. In all, we need

$$\sum_{i=1}^{k}\sum_{j=1}^{i} C + \sum_{i=k+1}^{n} kC = k(k+1)/2 + k(n-k) = \left(n+\frac{1}{2}\right)k - \frac{1}{2}k^2$$

In all it's roughly a $O(n^2)$ algorithm. □

Exercise 5 [Binomial Coefficient modulo 2] Suppose we are only interested in whether $\binom{n}{k}$ is even or odd, i.e., we want to compute $\binom{n}{k} \mod 2$. You could do this by computing $\binom{n}{k}$ using dynamic programming and then taking the result modulo 2. What is the running time? Would you say this algorithm is efficient? Why or why not?

Proof: The running time is exactly the same as Exer.4., not efficcient, we don't have to compute the value of $\binom{n}{k}$ at all.
Below is the python code.

```
def hasNtwoFactor(asking):
    res = 0
    while (asking % 2 == 0):
        res += 1
        asking = asking / 2
    return res

def isEvenOrOdd(n, k):
    numerator = 0
    dominator = 0
    for i in range(1, n + 1):
        numerator += hasNtwoFactor(i)
    for i in range(1, k + 1):
        dominator += hasNtwoFactor(i)
    for i in range(1, n - k + 1):
        dominator += hasNtwoFactor(i)
    if (numerator - dominator > 0):
        return bool(1)
    else:
        return bool(0)
```

it's an $O(n)$ algorithm, faster than dpCompute.

We also have another algorithm based on **Lucas Theorem**

```
def C(n, k):
  if n & k == k:
        return 1;
    else:
        return 0;
```

Now, let's proof this simple algorithm why it's correct.

Theorem 6 ***(Lucas Theorem)*** *For non-negative integers m and n and a prime p, the following congruence relation holds:*

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} (mod\ p),$$

where

$$m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0$$

, and

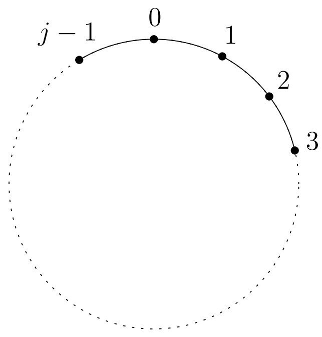
$$n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$$

are the base p expansions of m and n respectively. This uses the convention that $\binom{m}{n} = 0$ *if* $m < n$*.*

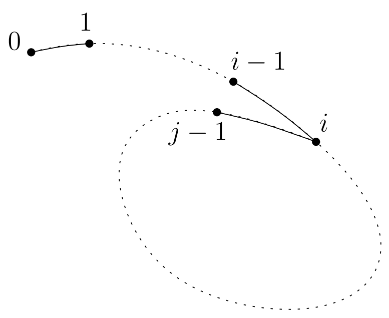
Here, we let p = 2, and we have $\binom{0}{0} = \binom{1}{0} = \binom{1}{1} = 1$ and $\binom{0}{1} = 0$. Hence, we get $\binom{n}{k} \equiv 1(mod\ 2)$ if and only if every bit of k is less than that of n in binary representation. That means $n\ and\ k = k$.

The time complexity of this algorithm is $\Theta(log(n))$ $\square$

Exercise 7 Remember the "period" algorithm for computing $F'_n := (F_n \bmod k)$ discussed in class: (1) find some i, j between 0 and k^2 for which $F'_i = F'_j$ and $F'_{i+1} = F'_{j+1}$. Then for $d := j - i$ the sequence F'_n will repeat every d steps, as there will be a cycle. This cycle can either be a "true cycle" or a "lasso":



a cycle



a lasso

Show that a lasso cannot happen. That is, show that the smallest i for which this happens is 0, i.e, for some j we have $F'_0 = F'_j$ and $F'_1 = F'_{j+1}$ and thus $F'_n = F'_{n \bmod j}$.

Proof: Consider a pair $g(i) = \left(F'_{i-1}, F'_i\right), i \geq 0$. We know that $\forall i, g(i) \in [0, n) \times [0, n)$ which is a finite set with k^2 members.

Consider the following set

$$A = \{g(1), g(2), \cdots, g\left(k^2\right), g\left(k^2+1\right)\}.$$

By **Pigeonhole Principle**, we know that $\exists r < s \in \{1, 2, \cdots, k^2+1\}$ such that $g(r) = g(s)$ which is

$$F'(r-1) = F'(s-1), \quad F'(r) = F'(s).$$

Since $F'(r-2)$ is determined by $F'(r) - F'(r-1)$, we can imply that

$$F'(0) = F'(s-r), \quad F'(1) = F'(s-r+1).$$

Hence this cycle is a "true cycle". □

Appendix

Pseudocode for Exercise 3, 4

Algorithm 1: A function f using recursive algorithm.

```
Input: input parameters $n, k$
Output: $\binom{n}{k}$
if $n = k$ or $k = 0$ then
  | return 1
end
else
  | return $f(n-1, k-1) + f(n-1, k)$
end
```

Algorithm 2: A function g using dynamic algorithm.

```
Input: input parameters $n, k$
Output: $\binom{n}{k}$
if $n = k$ or $k = 0$ then
  | return 1
end
else
  | return $f(n-1, k-1) + f(n-1, k)$
end
```