# CS217 – Algorithm Design and Analysis

# Homework 1

Not Strong Enough

March 8, 2020

**1**

Prove the more precise bound of the school method.

*Proof.* Assume $a$ has $n$ bits and $b$ has $k$ bits($k < n$), in the school method of division, calculate in rounds.

In each round, extend $b$ by filling zeros into lower bits and make $b$ have same bit size as $a$. If the extended $b$ is smaller than $a$, let $a$ minus $b$.

It is obvious that the first bit of $b$ is 1, so each minus will decrease the bit size of $a$. Besides, since the minus is made only when the extended $b$ is smaller than the current value of $a$, if the bit size of $a$ is smaller than $k$, the calculation will end. So there are at most $n - k$ rounds.

Now consider the minus. Since the lower bits of extended $b$ are 0, only the first $k$ bits of extended $b$ costs. So each minus has $k$ operations.

In all, there are at most $(n - k) * k$ operations, so the complexity is $O(k(n - k))$. □

**2**

Prove the complexity of Euclid algorithm is $O(n^2)$

*Proof.* In each round, the Euclid algorithm calculates $a\%b$(assume $a \geq b$) which is less than $b$. If the result is not 0, it uses the result with b to do the new round. So we can assume that, in each round, the two calculated numbers are:

$$(x_0, x_1), (x_1, x_2) \dots (x_{m-1}, x_m)$$

Where $x_0 > x_1 > \dots x_m, x_{m-1}\%x_m = 0$. Let the bit size of $x_i$ be $t_i$, then $t_0 = n, t_1 = k$, using the result of Exercise 1 we can find that the total operation number is:

$$\begin{aligned} O(t_1(t_0 - t_1)) + O(t_2(t_1 - t_2)) + \dots + O(t_m(t_{m-1} - t_m)) &\leq \sum_{i=1}^{m} t_i t_{i-1} - \sum_{i=1}^{m} t_i^2 && (1) \\ &< \sum_{i=0}^{m-1} t_i^2 - \sum_{i=1}^{m} t_i^2 && (2) \\ &= t_0^2 = n^2 && (3) \end{aligned}$$

So we can see the operation number of Euclid algorithm is $O(n^2)$ □

### 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?

**Algorithm 1** Binom_BF: A Recursive Algorithm for the Binomial Coefficient.

**if** $k = 0$ or $k = n$ **then**
  **return** 1
**else**
  **return** $Binom\_BF(n-1, k-1) + Binom\_BF(n-1, k)$
**end if**

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

In order to count the number of operations in the recursive algorithm to calculate $\binom{n}{k}$. We shall take a look at how the values are calculated. While recursively calculating Fibonacci number $F_n$ can be represented by a tree, we can show this process by a grid graph. A value $\binom{n}{k}$, in grid $(n, k)$, is obtained by $(n-1, k-1)$ and $(n-1, k)$. This indicates that there are edges from $(n-1, k-1)$ and $(n-1, k)$ to $(n, k)$.

The recursive algorithm actually tries all possible paths to $(n, k)$, starting at $(i, j)$ where $j = 0$ or $j = i$. That is to say, the time that value is merged as $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$, is the number of paths going through $(i, j)$, which, by the definition of binomial coefficients, is $\binom{n-i}{k-j}$.

Considering the complexity of integer addition $O(\log \text{ bitsize})$, we can get the running time of the recursive algorithm:

$$\sum_{i=0}^{n}\sum_{j=1}^{i-1}\binom{n-i}{k-j}\log\binom{i}{j} < \sum_{i=0}^{n}\sum_{j=1}^{i-1}\binom{n-i}{k-j}\log\binom{n}{\lfloor\frac{n}{2}\rfloor}.$$

Using the Stirling Formula, we have

$$O(\log\binom{2n}{n}) = O((2n+\frac{1}{2})\log 2n - (n+\frac{1}{2})\log n - (n+\frac{1}{2})\log n) = O(n).$$

The upper bound is

$$O(\sum_{i=0}^{n}\sum_{j=1}^{i-1}\binom{n-i}{k-j}\log\binom{n}{\lfloor\frac{n}{2}\rfloor}) = O(n\binom{n}{k}).$$

And the lower bound is

$$\Omega(\binom{n}{k}).$$

This algorithm is not efficient, because it cannot be done in polynomial time.

## 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?

**Algorithm 2** Caluculate Binomial Coefficient Using DP

Create a 2-dimension array $arr$
**for** $i = 0$ to $n$ **do**
  $arr[i][0] = 1$
**end for**
**for** $i = 0$ to $k$ **do**
  $arr[i][i] = 1$
**end for**
**for** $i = 1$ to $n$ **do**
  **for** $j = 1$ to $\min(i-1,k)$ **do**
    $arr[i][j] = arr[i-1][j] + arr[i-1][j-1]$
  **end for**
**end for**
**return** $arr[n][k]$

```
def calc_dp(n,k):
    arr = [[0 for i in range(k+1)] for j in range(n+1)]
    for i in range(n+1):
        arr[i][0]=1
    for i in range(k+1):
        arr[i][i]=1
    for i in range(1,n+1):
        for j in range(1,min(i-1,k)+1):
            arr[i][j]=arr[i-1][j]+arr[i-1][j-1]
return arr[n][k]
```

Complexity analysis:

When we traverse the array and visit `arr[i][j]`, we actually perform the add operation $\binom{i}{j} = \binom{i-1}{j} + \binom{i-1}{j-1}$. So the operation costs $O(\log \binom{i}{j})$ time. Using the Stirling Formula, we can estimate $\log \binom{i}{j} \approx (i+\frac{1}{2}) \log i - (i-j+\frac{1}{2}) \log(i-j) - (j+\frac{1}{2}) \log j = O(i)$. The number of nodes we visit is $O(k \cdot (2n-k)/2) = O(kn)$. Since we will only visit `arr[i][j]` once, we can estimate the total complexity as below: the upper bound is $O(kn \cdot n) = O(kn^2)$, the lower bound is $\Omega(kn)$. The algorithm is efficient, because it can be completed in polynomial time and there are no redundant operations.

Also, we can get the upper bound of the running time in another way:

$$\sum_{i=0}^{n} \sum_{j=0}^{\min(i,k)} \log \binom{i}{j} = (\sum_{i=0}^{k} \sum_{j=0}^{i} + \sum_{i=k+1}^{n} \sum_{j=0}^{k}) \log \binom{i}{j}.$$

We have

$$\begin{aligned}\sum_{i=0}^{k}\sum_{j=0}\binom{i}{j} &= \sum_{i=0}^{k}\sum_{j=0}^{i}(i\log i - j\log j - (i-j)\log(i-j)) \\ &= \sum_{i=0}^{k}(i^2\log i + (\frac{i^2}{4} - \frac{i^2}{2}\log i) + (\frac{i^2}{4} - \frac{i^2}{2}\log i)) \\ &= O(k^3).\end{aligned}$$

And

$$\begin{aligned}\sum_{i=k}^{n}\sum_{j=0}^{k} i\log i - j\log j - (i-j)\log(i-j) &= \sum_{i=k}^{n}(ki\log i - (\frac{k^2}{2}\log k - \frac{k^2}{4}) - (\frac{i^2}{2}\log i - \frac{i^2}{4}) + \frac{(i-k)^2}{2}\log(i-k) - \frac{(i-k)^2}{4}) \\ &= O(nk^2\log n).\end{aligned}$$

Thus the running time is $O(nk^2\log n)$.

**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?

The running time is the same as Problem 4. It's not efficient, because it cannot be done in polynomial time corresponding to the output size. However, we can do modulo operation after every addition, so that the complexity of addition can be reduced to $O(1)$, so that the total complexity can be $O(n \cdot k)$.

Furthermore, we can introduce a new algorithm to compute this problem(See Algorithm 3).

**Algorithm 3** Caluculate Binomial Coefficient Modulo 2

```
factor = an empty array
for i = 0 to n do
    factor[i] = calculateFactor2(i)
end for
for i = 1 to n do
    factor[i] = factor[i] + factor[i − 1]
end for
factorLeft = factor[n] − factor[n − k] − factor[k]
if factorLeft > 0 then
    return 0
else
    return 1
end if
```

**Algorithm 4** calculateFactor2

```
Require: the number i to be calculated
if i == 0 then
    return 0
end if
cnt = 0
while the lowest bit in i is 0 do
    i = i >> 1
    cnt = cnt + 1
end while
return cnt
```

The main idea is to calculate the number of factor 2 in $\binom{n}{k}$, equivalently, calculating the number of factor 2 in $n!$, $(n-k)!$ and $k!$, based on the fact that $\binom{n}{k} = \frac{n!}{(n-k)!\cdot k!}$.

The calculateFactor2(i) function is $O(\log i)$. The addition of `factor[i]` and `factor[i-1]` is also $O(\log i)$, because `factor[i]` represents the number of factor 2 in $i!$, which is $O(i)$. So the total complexity should be:

$$O(\sum_{i=1}^{n} \log i) = O(\log(n!)) = O(n \log n) \tag{4}$$

**6**

Remember the "period" algorithm for computing $F'_n := (F_n \mod k)$ 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". Show that a lasso cannot happen. That is, show that the smallest $i$ for which this happens is 0.

*Proof.* We prove it by contradiction.

Let $i_0 > 0$ be the smallest $i$ for which this happen, i.e, for some $j > 0$ we have: for $\forall p \geqslant i_0 > 0, F'_{p+j} = F'_p$. Since for $p \geqslant 1$, the fibonacci numbers are defined as $F_{p+1} = F_p + F_{p-1}$, we have $F'_{p+1} = (F'_p + F'_{p-1}) \mod k$. It follows that $F'_{p-1} = (F'_{p+1} - F'_p + k) \mod k$.

Because $F'_{i_0} = F'_{i_0+j}, F'_{i_0+1} = F'_{i_0+1+j}$, we have

$$\begin{aligned} F'_{i_0-1} &= (F'_{i_0+1} - F'_{i_0} + k) \mod k \\ &= (F'_{i_0+1+j} - F'_{i_0+j} + k) \mod k \\ &= F'_{i_0-1+j} \end{aligned}$$

, which is contradict with out hypothesis.

Thus, $i_0$ isn't the smallest $i$, and we can conclude that the smallest $i$ is 0. □