Homework 1

Ji Jiabao

2020 年 3 月 7 日

Exer.1:

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.

Exer.2:

We prove it by induction on bits of a, b, marked as n, m, and for r in each iteration, we have n
suppose it takes k times of iteration.

$$T = \sum_{i=1}^{k} T_i = \sum_{i=1}^{k} n_i * m_i < \sum_{i=1}^{k} n_i * (n_i - n_{i-1})$$

Exer.3:

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

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

Exer.4:

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

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

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

Exer.5:

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.