# Introduction to Bioinformatics Algorithms
# Homework 2

Saad Mneimneh
Computer Science
Hunter College of CUNY

**Problem 1: Coin Change**
(a) The greedy algorithm for coin change can be described as:

$$G(n) = 1 + G(n - c)$$

where $c$ is the largest coin value less or equal to $n$.

$G(n)$
  **if** $n > 0$
    **then** let $c$ be largest coin value $\leq n$
        **return** $1 + G(n - c)$
    **else return** 0

Transform this algorithm into a dynamic programming algorithm to compute $G(0), G(1), \ldots, G(n)$. What is the running time of your algorithm?

(b) Describe a dynamic programming algorithm to solve the coin change problem in general. What is the running time of your algorithm?

(c) Do some research online for an algorithm that determines, given denominations $c_1 > c_2 > \ldots > c_d = 1$, whether the greedy coin change algorithm works correctly. In particular, you may consider: `http://www.cs.cornell.edu/~kozen/papers/change.pdf`. Implement the algorithm you find.

**Problem 2: Fibonacci Revisited**
Consider the following algorithm:

$fib(a, b, n)$
  while $n > 0$
      $b \leftarrow a + b$
      $a \leftarrow b - a$
      $n \leftarrow n - 1$
  return $a$

$fib(n)$
  return $fib(0, 1, n)$

This algorithm requires $O(n)$ arithmetic operations. However, since Fibonacci numbers grow fast, it is not reasonable to assume that arithmetic operations take constant time. Addition of $b$ bit numbers take $O(b)$ time (standard addition algorithm we do by hand). One can show that the $n^{\text{th}}$ Fibonacci number

is $\Theta(\phi^n)$; therefore, all intermediate results while computing $fib(n)$ need $O(n)$ bits. This makes the above algorithm an $O(n^2)$ time algorithm. We can do better.

Consider the matrix:

$$F = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

It is easy to verify that $F^n_{1,2}$ is the $n$th Fibonacci number. This means, we only need to multiply the matrix by itself $n$ times, each matrix multiplication involves 8 multiplications and 4 additions. The bottleneck is multiplication. Assume that we can multiply two $n$ bit numbers is $O(n^\alpha)$ time for $1 < \alpha < 2$. Then the total running time of this algorithm will be $O(n^{1+\alpha})$, not an improvement over the $O(n^2)$ bound.

However, we can use a technique called repeated squaring. Consider the function *pow* (for power).

```
pow(F, i) (compute F^i)
    if i > 0
      then if i is even
              then return square(pow(F, i/2))
              else return F × pow(F, i − 1)
      else return identity matrix
```

(a) What is the number of multiplications performed by this algorithm using Big-O notation?

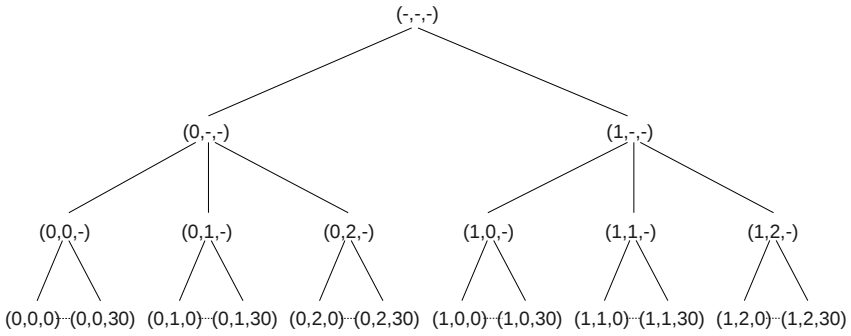
(b) Show that $\log^a n = o(n^\epsilon)$ for any positive $a$ and $\epsilon$, i.e. $\lim_{n\to\infty} \frac{\log^a n}{n^\epsilon} = 0$. This is small $o$ notation and it means that that $\log^a n < cn^\epsilon$ for **every** constant $c > 0$ (not just some constant $c$) and large enough $n$.

(c) Describe a better than $O(n^2)$ algorithm for computing the $n$th Fibonacci number.

**Problem 3: Enumeration**

In the previous homework, we considered the enumeration from $(0, 0, \ldots, 0)$ to $(n_1, n_2, \ldots, n_d)$. We now explore this enumeration using a tree structure.

For example, assume $d = 3$ and $n_1 = 1$, $n_2 = 2$, and $n_3 = 30$. Here's a tree structure that shows all possible counts as the leaves.




The preorder traversal of the tree gives internal nodes and leaves of the form $(a[1], a[2], a[3])$:

$$(-,-,-),(0,-,-),(0,0,-),(0,0,0),\ldots,(0,0,30),(0,1,-),(0,1,0),\ldots,(0,1,30),$$

$$(0,2,-),(0,2,0),\ldots,(0,2,30),(1,-,-),\ldots\ldots\ldots,(1,2,30),(-,-,-)$$

Let $a[0]$ encode the level in the tree, i.e. this is also the number of values not equal to $'-'$ in $a$. Therefore, if $a[0] = l$, $a[l+1], \ldots a[d]$ are all $'-'$ (as far as implementation is concerned, they can be ignored).

(a) Assuming $n_i \neq 0$ for all $i$, show that the number of nodes in the tree is $\Theta(n_1 \times n_2 \times \ldots \times n_d)$ for a fixed $d$.

(b) Given $a$, write a function $next(a, n, d)$, where $n$ contains $n_1, n_2, \ldots, n_d$, that changes $a$ to the next node in the preorder traversal of the tree. Do not explicitly build the tree structure, just manipulate the values in $a$. Test your code by generating all the nodes of the tree in the preorder traversal.