Introduction to Bioinformatics Algorithms
Homework 1 Solution

Saad Mneimneh
Computer Science
Hunter College of CUNY

Problem 1: Coin Change
Write a function that takes an integer d, an array c, where $c[1] > c[2] > \ldots > c[d] = 1$, an integer n, and an array k, and performs the greedy coin change problem to make n. Therefore, it should modify k such that:

$$c[1]k[1] + c[2]k[2] + \ldots + c[d]k[d] = n$$

Also make your function return $\sum_{i=1}^{d} k[i]$, which is the total number of coins used.

Solution: Here's a pseudocode for the greedy algorithm, as described in class.

coin_greedy(n, c, k, d)
  $num \leftarrow 0$
  for $i \leftarrow 1$ to d
    $k[i] \leftarrow n/c[i]$ ▷ integer division
    $n \leftarrow n - c[i]k[i]$
    $num \leftarrow num + k[i]$
  return num

If indexing starts at 0, then i should iterate from 0 to $d - 1$ instead.

Problem 2: Exhaustive enumeration
Write two algorithms that iterate over every index from $(0, 0, \ldots, 0)$ to $(n_1, n_2, \ldots n_d)$. Make one algorithm recursive and one iterative.

Solution: Here's a pseudocode for advancing the count to the next. Repeated use of this can iterate over all of them. For convenience, I will make the function return a boolean to indicate whether we were able to increment a position or not.

advance_rec(a, d, n)
  if $d > 0$ and $a[d] = n[d]$ ▷ this position reached the max
    then $a[d] \leftarrow 0$ ▷ reset and recurse
      return advance_rec($a, d - 1, n$)
    else if $d = 0$ ▷ all positions have been reset
      then return false
      else $a[d] \leftarrow a[d] + 1$ ▷ $a[d] \neq n[d]$, so increment
        return true

If indexing starts at 0, then $a[d]$ should be replaced by $a[d-1]$. Since we have a tail recursion in the form presented in class, it can be eliminated using the standard technique we discussed, i.e. (1) replacing the if by a while, (2) changing the recursive call to an update of parameters, and finally (3) removing the else if any.

```
advance_iter(a, d, n)
  while d > 0 and a[d] = c[d]     ▷ this position reached the max
     a[d] ← 0     ▷ reset and iterate
     d ← d − 1
  if d = 0      ▷ all positions have been reset
    then return false
    else a[d] ← a[d] + 1     ▷ a[d] ≠ n[d], so increment
         return true
```

Assuming we initialize a to $(0, 0, \ldots, 0)$, both versions can be used as follows:

```
enumerate_rec(a, d, n)
  do something with a, e.g. output a
  if advance(a, d, n)
    enumerate_rec(a, d, n)
```

or

```
enumerate_iter(a, d, n)
  repeat
    do something with a, e.g. output a
  until advance(a, d, n)
```

Problem 3: Rabbits with limited life span
Modify the Fibonacci sequence by making every pair of rabbits die after giving birth to their k^{th} pair (assume $k \geq 1$). Your program should output F_n given n and k. Investigate the growth of the sequence by exploring several values of k.

Solution: We can keep track of the number of adult and newborn pairs in each time step. For any given time step n, $fib(n) = adult_n + newborn_n$. We also know that these numbers evolve as follows:

$$adult_n \leftarrow adult_{n-1} + newborn_{n-1}$$

$$newborn_n \leftarrow adult_{n-1}$$

This will give the original Fibonacci sequence (I am assuming $fib(0) = 0$ and $fib(1) = 1$).

```
fib(n)
 if n ≤ 1
   then return n
 adult ← 0
 newborn ← 1
 for i ← 2 to n
   adult ← adult + newborn
   newborn ← adult − newborn
 return adult + newborn
```

The modification for limited life span will have to recall at time n, the value of $newborn_{n-1}$ (these will become new adults at time n) and, therefore, subtract that number from the total at time $n+k$. This can be achieved by a queue of length k in the following way: at time n, we insert $newborn_{n-1}$. An insertion at time n, will have to drop the element inserted at time $n-k$, since the queue has length k only. If the dropped element is returned, that's the value we have to subtract. Therefore, let us assume the existence of a function insert(a) that inserts a into a queue, and drops and returns the value inserted k steps before, or returns 0 if none.

```
fib(n)
 if n ≤ 1
   then return n
 adult ← 0
 newborn ← 1
 for i ← 2 to n
   dropped ← insert(newborn)
   adult ← adult + newborn
   newborn ← adult − newborn
   adults ← adults − dropped
 return adult + newborn
```

This functionality can be achieved by a circular array of size k, indexed from 0 to $k-1$

```
init
  for i ← 0 to k − 1
    queue[i] ← 0
  pos ← 0

insert(a)
  dropped ← queue[pos]
  queue[pos] ← a
  pos ← (pos + 1) mod k
```

Trying for several values of k reveals that for $n > k+1$, $fib(n) = \sum_{i=n-1-k}^{n-2} fib(i)$ (k terms), and $fib(n)$ remains unchanged for $n \leq k+1$ (if we assume $fib(0) = fib(1) = 1$, thus shifting the sequence by 1, then the threshold $k+1$ is changed to k, code below).

```
fib(n)
 if n = 0
   then return 1
 adult ← 0
 newborn ← 1
 for i ← 1 to n
   dropped ← insert(newborn)
   adult ← adult + newborn
   newborn ← adult − newborn
   adults ← adults − dropped
 return adult + newborn
```