Introduction to Bioinformatics Algorithms
Homework 2 Solution

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?

Solution: I am going to assume here that c is indexed from 0 to $d - 1$ with $c[0] = 1$.

GreedyDP(c, d, n, G, bt)
  $G[0] \leftarrow 0$ ▷ initialization
  $largestfit \leftarrow 0$ ▷ largest coin we can fit now is $c[0]$=1
  for $i \leftarrow 1$ to n
    while $largestfit + 1 < d$ and $c[largestfit + 1] <= i$ ▷ can be replaced by if
      $largestfit \leftarrow largestfit + 1$
    $G[i] \leftarrow 1 + G[i - c[largestfit]]$
    if $bt \neq 0$ ▷ do you want backtracking?
      then $backtrack[i] \leftarrow largestfit$

It is easy to show that advancing the largest fit needs to only check the next largest coin (that's why the while can be replaced by an if). Therefore, the amount of work done per iteration is constant, i.e. $O(1)$. The running time of the algorithm is $O(n)$.

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

Solution: As discussed in class, here we need to use the best fit coin instead of the largest fit.

```
generalDP(c, d, n, G, bt)
  G[0] ← 0 ▷ initialization
  largestfit ← 0 ▷ largest coin we can fit now is c[0]=1
  for i ← 1 to n
      while largestfit + 1 < d and c[largestfit + 1] <= i ▷ can be replaced by if
         largestfit ← largestfit + 1
      min ← i
      bestfit ← largestfit ▷ just arbitrary choice
      for j ← 0 to largestfit
          if 1 + G[i − c[j]] < min
            then min ← 1 + G[i − c[j]]
                 bestfit = j
      G[i] ← 1 + G[i − c[bestfit]]
      if bt ≠ 0 ▷ do you want backtracking?
        then backtrack[i] ← bestfit
```

Now all coin values less than or equal to the largest fit must be checked in each iteration, for a running time of $O(dn)$.

(c) Do some research online for an algorithm that determines, given denominations $c_0 = 1 < c_1 < \ldots < c_{d-1}$, 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.

Solution: A coin system is *canonical* (always works with Greedy) iff there is no value i in the range $c_2 + 1 < i < c_{d-2} + c_{d-1}$ such that $G[i] > 1 + G[i - c[j]]$ for some j. Here's a simple implementation of this idea:

```
canonical(c, d)
  if d < 3
    then return true ▷ every 3 coin system is canonical
  GreedyDP(c, d, c[d − 2] + c[d − 1] − 1, G, 0)
  for i ← c[2] + 2 to c[d − 2] + c[d − 1]
      for j ← 0 to d − 1
          if i ≥ c[j] and G[i] > 1 + G[i − c[j]]
            then return false
  return true
```

The nested loop is $O(d)$ time. The outer loop and the greedy algorithm are both upper bounded by $O(c_{d-1} + c_{d-2}) = O(2c_{d-1}) = O(c_{d-1})$. So the total running time is $O(dc_{d-1})$, this is the number of coins multiplied by the largest coin value.

Problem 2: Fibonacci Revisited
Consider the following algorithm:

```
fib(a, b, n)
   while n > 0
         b ← a + b
         a ← b − a
         n ← 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^{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 nth 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?

Solution: The power i is divided by 2 when it's even. If this happens repeatedly, i.e. when i is a power of 2, then we have exactly $\log_2 i + 1$ multiplications, which is $O(\log i)$. Since when i is odd $i-1$ is even, the number of multiplications is at most doubled. This is still $O(\log i)$.

(b) Show that $\log^a n = o(n^\epsilon)$ for any positive a and ϵ, 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.

Solution: Using l'Hospital's rule:

$$\lim_{n\to\infty} \frac{\log^a n}{n^\epsilon} = \lim_{n\to\infty} \frac{a \log^{a-1} n(1/n)}{\epsilon n^{\epsilon-1}} = \lim_{n\to\infty} \frac{a}{\epsilon} \frac{\log^{a-1} n}{n^\epsilon}$$

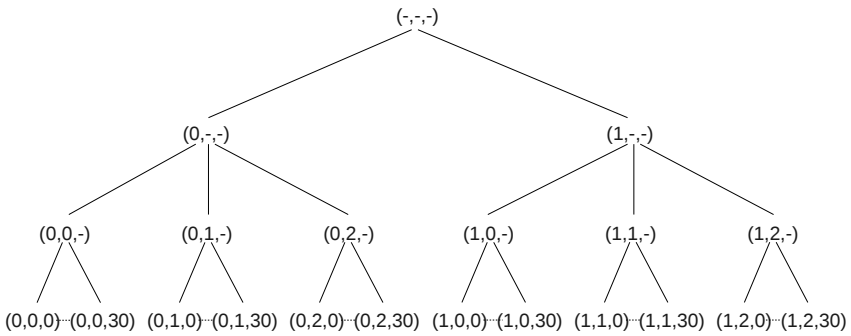
This can be repeated until the power of $\log n$ is negative and the limit is 0.

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

Using part (a), we can compute F^n using $O(\log n)$ multiplications. Each multiplication takes $O(n^\alpha)$ time, where $\alpha < 2$. Therefore, the running time is $O(n^\alpha \log n) = o(n^2)$, because $\lim_{n\to\infty} \frac{n^\alpha \log n}{n^2} = \lim_{n\to\infty} \frac{\log n}{n^{2-\alpha}} = 0$

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.

Solution: It is easy to show that the number of leaves is equal to $L = (n_1 + 1)(n_2 + 1)\ldots(n_d + 1)$. If the tree has N nodes (including the leaves), then definitely $N \geq L$. In addition, since $n_i > 0$, we can also show that the number of nodes in level l is at most half the number of nodes in level $l + 1$. Therefore, the total number of nodes is upper bounded by $L(1 + 1/2 + 1/4 + \ldots) \leq 2L$. So $N = \Theta(L)$. This shows that the complexity of generating nodes in the tree is comparable to simply enumerating the leaves. We now show that $L = \Theta(n_1 n_2 \ldots n_d)$.
It is obvious that $L > n_1 n_2 \ldots n_d$. So all we need to show is that $L < c n_1 n_2 \ldots n_d$ for some constant $c > 0$. Well, $n_i + 1 \leq 2n_i$. So $L \leq 2^d n_1 n_2 \ldots n_d$. Finally, $L = \Theta(n_1 n_2 \ldots n_d)$. Since $N = \Theta(L)$ and $L = \Theta(n_1 n_2 \ldots n_d)$, then $N = \Theta(n_1 n_2 \ldots n_d)$.

(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.

Solution: If a is not a leaf, say a has level $a[0] = l < d$, then we simply replace the first $-$ by a 0 and increase the level by 1. This can be done by the following: $a[0] \leftarrow a[0] + 1$ and $a[a[0]] \leftarrow 0$. On the other hand, if $a[0] = d$, then we need to check $a[d]$. If $a[d] < n_d$, we simply increase it by 1. Otherwise, if $a[d] = n_d$, then we go up one level by decreasing $a[0]$, and check $a[d-1]$. We do this repeatedly until we find some l where $a[l] < n_l$ and increase $a[l]$ by 1. If we ever set $a[0] = 0$, then we simply stop because there is no successor (we go back to root).

```
▷ we assume that a has been initialized such that a[0] = 0
next(a, n, d)
  if a[0] < d
    then a[0] ← a[0] + 1
         a[a[0]] = 0
         return
  for i ← d downto 1
      if a[i] < n[i]
        then a[i] ← a[i] + 1
             a[0] ← i
             return
  a[0] ← 0
```