CSCI 3100: Algorithms
Homework 3

Required Problems

1. Consider a graph with n vertices, where each vertex has some real valued weight. Recall that a subset of the vertices is called *independent* if no two of them are joined by an edge. Finding large independent sets is difficult in general, as we will discuss later this semester in class, but can be done on some simple classes of graphs.

   Call a graph a *path* if its vertices can be written as $v_1, v_2, \ldots, v_n$ with an edge between each v_i and v_{i+1} (but no other edges). With each vertex v_i, we associate a weight w_i.

   Our goal in this problem is to find the largest weight independent set. (Note that this is different from the largest independent set, since here we take the weights into account!)

   (a) Construct an example showing why the following simple *greedy* algorithm does NOT always work.

   ```
   S ← ∅
   While G is not empty:
      Pick a node v_i of maximum weight
      Add v_i to S
      Delete v_i and its neighbors from G
   Return S
   ```

   (b) Construct an example showing why the following different simple *greedy* algorithm does NOT always work.

   ```
   S_1 ← {v_i with i odd}
   S_2 ← {v_i with i even}
   oddsum ← sum of all weights in S_1
   evensum ← sum of all weights in S_2
   if evensum > oddsum
      return S_2
   else
      return S_1
   ```

   (c) Give an algorithm that takes an n-vertex path G with weights and returns an independent set of maximum total weight. Your running time should be polynomial in n. (Hint: Yes, a path is always a tree. But I want you to adapt that algorithm - your data structure will be much simpler, but must take weights into account! If you understand dynamic programming on trees, this will be even simpler.)

2. The residents of the last city on Earth, Zion, must defend themselves from an onslaught of killer flying robots. (Yes, you may have seen this movie.). While they traditionally utilize heavy artillery and kung fu, their newest and most effective method of defense is setting off an EMP to disable incoming robots. They must design an efficient and optimal algorithm to decide when the EMP goes off, so as to kill as many robots as possible as they come in the front gate; however, this is made more complex by the fact that their EMP must be charged, and it gets stronger the longer it charges.

We formalize this in the following way:

- Every second (from 1 to n), some number of robots x_i arrives at the gate, where they can be reached by the EMP if they set it off. Since they have advanced remote sensing techniques, they know all n values in advance.
- The EMP's charge is described by a function; if it's been charging for k seconds, it's capable of killing $charge(k)$ or x_k robots, whichever is smaller (since you can only kill as many as are actually arriving). We'll assume the EMP begins completely drained, so that if it goes off for the first time at second i in the attack, it will kill $\min\{charge(i), x_i\}$ robots; from then on, if it was set off at time i and then is next set off at time j, it kills $\min\{charge(j-i), x_j\}$ robots.

Your goal is to design an algorithm that, given the arrivals $x_1, x_2, \ldots x_n$ and the function $charge$, chooses the times to fire the EMP which kill as many robots as possible. (Be sure to justify correctness as well as runtime and space!)

Example: Suppose $n = 4$, and we have $x_1 = 1$, $x_2 = 10$, $x_3 = 10$, and $x_4 = 1$, with $charge(i) = 2^{i-1}$. Then the best solution is to activate at times 3 and 4; at time 3, it will kill $\min 2^{3-1}, 10 = 4$ robots, and then in the 4^{th} second it will kill 1 more robot, for a total of 4 robots. (You can check that any other solution will not kill as many.)

3. Problem 47 from Chapter 3 of the textbook.

4. Extra Credit: Problem 17 from Chapter 3 of the textbook: go earn style points in vogue vogue revolution! (Hint: Part a is much easier than it looks. What's the easiest, or laziest, strategy you can think of? Part b will take some dynamic programming, though.)

   Describe and analyze an algorithm to decide, given 3 strings X, Y, and Z, whether Z is a smooth shuffle of X and Y.

5. Sample Solved Problem: A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways:
   BANANAANANAS, BANANAANANAS, or BANANANANAS.

   Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:
   PRODGYRNAM AMMIINCG and DYPRONGARMAMMICING.

   Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

   Solution: We define a boolean function $Shuf(i, j)$, which is True if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

   - Shuf(i, j) = true if $i = j = 0$
   - Shuf$(0, j-1)$ AND $(B[j] = C[j])$ if $i = 0$ and $j > 0$

- Shuf$(i-1,0)$ AND $(A[i]=C[i])$ if $i>0$ and $j=0$
- (Shuf$(i-1,j)$ AND $(A[i]=C[i+j])$) OR (Shuf$(i,j-1)$ AND $(B[j]=C[i+j])$) if $i>0$ and $j>0$

The proof that this formulation is correct can be shown via induction: if you're considering the $i+j^{th}$ character of C, it must be from either $A[i]$ or $B[j]$. We are trying both options, and returning true if either works. The base cases handle either A or B being empty, in which case either we've matched everything (and both are 0) or we must exactly match the rest of C to which ever string is left.

We need to compute Shuf(m, n). We can memoize all function values into a two-dimensional array Shuf[0 .. m][0 .. n]. Each array entry Shuf[i, j] depends only on the entries immediately below and immediately to the right: Shuf[i-1, j] and Shuf[i, j-1]. Thus, we can fill the array in standard row-major order.

```
SHUFFLE?(A[1..m], B[1..n], C[1..m+n]):
  Shuf[0,0] ← TRUE
  for j ← 1 to n
      Shuf[0,j] ← Shuf[0,j−1] ∧ (B[j] = C[j])
  for i ← 1 to n
      Shuf[i,0] ← Shuf[i−1,0] ∧ (A[i] = B[i])
      for j ← 1 to n
          Shuf[i,j] ← FALSE
          if A[i] = C[i+j]
              Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i−1,j]
          if B[i] = C[i+j]
              Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i,j−1]
  return Shuf[m,n]
```

The algorithm runs in $O(mn)$ time, and (if we keep the entire 2d arrays) takes the same amount of space. This can be improved to $O(m)$ (or $O(n)$) by keeping only two rows: the one we are currently filling in, and the row immediately preceding it.

■