CS 4820, Spring 2018	**Homework 3, Problem 2**
Name: Rongguang Wang	
NetID: rw564	
Collaborators: Siyuao Liu (sl2928); Yihao Chen (yc2288)	

(2) In class we've been talking about applications of dynamic programming to optimization. There are also many applications of dynamic programming to counting, and to calculating probabilities. This exercise explores one such application. Recall that an instance of the *interval scheduling* problem consists of n intervals $I_1, I_2, \dots, I_n$, where each interval I_k (for $k = 1, \dots, n$) is a closed interval $[s_k, f_k]$ with start time s_k and finish time $f_k > s_k$. A set of intervals is *non-conflicting* if no two of its elments overlap.

In this exercise we assume we are given an instance of the interval scheduling problem such that the numbers $s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$ are all distinct, and such that the intervals are ordered by increasing finish time: $f_1 < f_2 < \cdots < f_n$.

(2a) *(7 points)*
Design an algorithm to count how many subsets of $\{I_1, I_2, \dots, I_n\}$ are non-conflicting. Remember that the empty set and one-element sets are always non-conflicting.

Solution:

The problem is aiming to design an algorithm to count the non-conflicting subset in a interval set. This is a typical dynamic programming problem which is similar to the Maximum Weight Interval Schedule ($MWIS$) problem. Assume there are intervals I from 0 to n in a set, the last interval should be consider first. The last interval I_n can be involved inside the subset or dropped. The total count of the possible subset including the situations that both include the last interval I_n or not include it. Let Count of the Non-conflicting Subset (CNS) be the iterative function for the dynamic programming and build an array called $p(n)$ to be the dynamic programming look-up table (DPT) which stores the maximum index of the interval that does not conflict with interval I_n. Thus, the series of intervals $I_{P(n)+1}$ *to* I_{n-1} conflicts with I_n. The recursive function $CNS(n) = CNS(p(n)) + CNS(n-1)$ which means that the count of the non-conflicting interval subsets in a set with the last interval n is equals to the count that include I_n and the count that does not include I_n. The count that includes I_n the last interval by calling the function $CNS(p(n))$ and the count that include I_n by calling the function $CNS(n-1)$. The function $CNS(n)$ will be called recursively until the input of all $CNS(n)$ becomes 0, which means the result becomes a branch of summation of $CNS(0)$. It is known that the count of non-conflicting interval subset of 0 is empty. Thus, $CNS(0) = 1$. Besides, pre-processing is needed in order to build the array $p(k)$ which requires sorting the starting and finishing time of each interval in increasing order. $p(k)$ is the largest index such that $f_{p(k)} < s_k$. If no such index exists, $p(k) = 0$.

Algorithm:
Assume the intervals in the set are numbered from 1 to n.
1) Sort the starting and finishing time of the intervals $\{s_1, ..., s_n, f_1, ..., f_n\}$ in increasing order
2) Build the array $p(k)$ based on the starting time and finishing time from last step which meets the requirement that $f_{p(k)} < s_k$

3) Call the function $CNS(n)$
$CNS(n)$:

```
if n = 0, return 1 (base case)
while n > 0
    return CNS(p(n)) + CNS(n-1)
end while
```

Algorithm's Correctness:

Theres an observation that the optimal solution to this problem either contains interval n, or it does not. An optimal solution that contains interval n consists of $\{interval\ n\} \cup \{non-conflicting\ subset\ of\ the\ intervals\ that\ finish\ before\ s_n\}$Let S be a non-conflicting set of intervals that contains interval n. So, $S = \{n\} \cup S'$. Then, every interval in S' does not conflict with interval n (either starts before s_n or finish after f_n). Every interval in S' starts before s_n and does not conflicts with interval n, hence, finishes before s_n. The count of the non-conflicting intervals in the set should include the count number both contains interval n and does not contains interval n. A recursive algorithm is suggested for this case study which is the function $CNS(n)$ described above. The function $CNS(n)$ is calculated by adding the count of count number both contains interval n and does not contains interval n.

Running Time Analysis:

The time spend in this algorithm consists of: sorting the starting and finishing time, build the array $p(k)$ and $CNS(n)$ function. The cost of sorting is $O(n \log n)$ time. Creating the array needs $O(n)$ time. As for the running time of the recursive algorithm $CNS(n)$, there are $(n + 1)$ nodes with constant time work each node which is $O(n)$. In total, the time needed is $O(n) + O(n \log n) = O(n \log n)$.

(2b) *(3 points)*
Let Ω denote the collection of all non-conflicting subsets of $\{I_1, \dots, I_n\}$. Given the list of intervals $I_1, \dots, I_n$, and an index k in the range $1 \leq k \leq n$, design an algorithm to compute the probability that a uniformly random element of Ω contains interval I_k.

In your solution to (2b), you may omit the running time analysis. The algorithm you design must still have running time bounded by a polynomial function of n, but you don't need to include the analysis of running time in your write-up. You are also free to use the algorithm from part (2a) as a subroutine in part (2b), even if you didn't succeed in solving (2a).

Solution:

The problem is aiming to find an algorithm which can compute the probability of that an interval I_k is belongs to a uniformly distributed set Ω which contains several collection of non-conflicting intervals. Given that $\Omega \subseteq S$ with $S = \{I_1, ..., I_n\}$. The probability of interval I_k that belongs to Ω is equals to the count of subset in Ω that contains I_k over the count of subset in Ω. The count of the subsets can be computed by using the $CNS(n)$ function from 2(a). Therefore, $P_r(I_k \in \Omega) = \frac{CNS(I_k \in \Omega)}{CNS(\Omega)}$. I_k is the interval that may belongs to some subset of Ω which can represented as $(A\ I_k\ B)$ where A is the set of intervals that finished before the stating time of I_k and B is the set of intervals that started after the finishing time of I_k. A and B can be represented as $A \subseteq (I_1, ..., I_{p(k)})$ and $B \subseteq (I_{q(k)}, ..., I_{p(k)})$ where $p(k)$ is the largest index such that $f_{p(k)} < s_k$ and $q(k)$ is the smallest index such that $s_j > f_k$. The count of the subsets that contains I_k is the combination of the count of the subset A and the count of subset B. Thus, the probability can be represented as $P_r(I_k \in \Omega) = \frac{CNS(I_k \in \Omega)}{CNS(\Omega)} = \frac{CNS(A) \cdot CNS(B)}{CNS(\Omega)}$. It is worth to clarify that the CNS function for $CNS(A)$ and $CNS(\Omega)$ is different from $CNS(B)$, since the total interval numbers are different. The intervals corresponding to $CNS(A)$ and $CNS(\Omega)$ is from 1 to n but the intervals corresponding to $CNS(B)$ is from $q(k)$ to n.

Algorithm:
1) Sort the starting and finishing time of the intervals $\{s_1, ..., s_n, f_1, ..., f_n\}$ in increasing order
2) Build the array $p(k)$ based on the starting time and finishing time from last step which meets the requirement that $f_{p(k)} < s_k$
3) Call $CNS(n)$ and $CNS(p(k))$
4) Build the array $q(k)$ based on the starting time and finishing time from the first step which meets the requirement that $s_j > f_k$
5) Drop the interval $I_{q(k)}$ and the intervals appear after $I_{q(k)}$
6) Call $CNS(n - q(k) + 1)$
7) Compute $\frac{CNS(n) \cdot CNS(p(k))}{CNS(n - q(k) + 1)}$

Algorithm's Correctness:

An exhaustive case analysis justifying correctness was already presented above.