⭐ Unless specified, you should justify your algorithm with proof of correctness and time complexity

CS180 Homework 4

Due: 11:59 pm PST, June 2 (No late submission is allowed)

1. (25 pt) You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i, you're presented with x_i terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

   You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s_1 terabytes, on the second day after a reboot, you can process s_2 terabytes, and so on, up to s_n; we assume $s_1 > s_2 > s_3 > \cdots > s_n > 0$. (Of course, on day i you can only process up to x_i terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

   The problem. Given the amounts of available data $x_1, x_2, ..., x_n$ for the next n days, and given the profile of your system as expressed by $s_1, s_2, ..., s_n$ (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process.

   Example. Suppose n = 4, and the values of x_i and s_i are given by the following table.

	day1	day2	day3	day4
x	10	1	7	7
s	8	4	2	1

   The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process $8 + 1 + 2 + 1 = 12$; and other rebooting strategies give you less than 19 as well.)

   (a) Give an example of an instance with the following properties.

   - There is a "surplus" of data in the sense that $x_i > s_1$ for every i.
   - The optimal solution reboots the system at least twice.

   In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

> s_i: the number of terabytes that can be process on i-th day after a reboot. $(s_1 > s_2 > ... > s_n > 0)$
>
> x_i: the maxmium terabytes that can be process on i-th day.
>
> Note : The system cannot process any data if reboot on that day.

Example:

Day	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
x	26	7	26	4	26	10	26
s	25	6	5	4	3	2	1

The optimal solution: reboot on day 2, 4, 6, thereby processing 25+0+25+0+25+0+25 = 100 terabytes.

(b) Give an efficient algorithm that takes values for $x_1, x_2, ..., x_n$ and $s_1, s_2, ..., s_n$ and returns the total number of terabytes processed by an optimal solution.

The question is in optimal structure. $OPT(i, j)$ = the maximum terabytes that can be processed starting from day i through day n to be , and the last reboot happened j days period befor day i.

Input:

- [x_1, x_2, ..., x_n] : x_i represents the maxmium terabytes that can be process on i-th day.
- [s_1, s_2, ..., s_n] : s_i represents the number of terabytes that can be process on i-th day after a reboot. $(s_1 > s_2 > ... > s_n > 0)$

Output: $Opt(i, j)$, a integer data type, optimal amout of terabytes that we can process

```
optimal_terabyte([x_1, x_2, ...., x_n], [s_1, s_2, ..., s_n]):
  for j in range (1, n):
    OPT(n, j) = min{x_n, s_j}
  Endfor
  for i in reversed(range(n)):     //i = n-1 -> i = 1
    for j in range(1, i):
      OPT(i,j) = max{OPT(i+1, 1), min(x_i, s_j) + OPT(i+1, j+1)}  //Take the larger one
    Endfor
  Endfor
  return OPT(1,1)
```

Time Complexity: Only $O(n^2)$ values are calcualted, each takes $O(1)$ time. Therefore, the total time for this algorithm takes $O(n^2)$ time.

Correctness Proof (by induction):

Base Case: Assume n = 1, which mean there is only one day to process data, Obviously, reboot is not the optimal solution since it cannot bring any benefits in this case. So $OPT(1, 1)$ = $min(x_1, s_1)$ is the correct answer. Hence the algorithm returns the correct answer.

IH: Assume n-1 > 1 days, the algorithm returns the total number of terabytes prcessesd by an optimal solution for n-1 days.

Induction Steps: Add one more day. Now we need to prove the IH works for n days period.

There are two cases on each of n days expect day n:

- Case 1 : Reboots the system

  The system cannot process anything (0 terabyte) on day i if reboot, and day (i+1) is the first day after reboot, j = 1 .

$$OPT(i, j) = 0 + OPT(i + 1, 1) = OPT(i + 1, 1)$$

- Case 2: Not reboot the system:

On day i you can process the minimum data of x_i and s_j.

$$OPT(i,j) = min\{x_i, s_j\} + OPT(i+1, j+1)$$

Finally, on the last day which is day n, reboot options cannot bring any benefits, it must not the optimal solution/. So

$$OPT(n,j) = min(x_n, s_j) \quad \text{for all } j \in [1,n]$$

Therefore, $OPT(i,j) = max\{OPT(i+1,1), min\{x_i + s_j\} + OPT(i+1, j+1)\}$

The algorithm holds this case.

2. (25 pt) A palindrome is a string that reads the same from left to right and from right to left. Design an algorithm to find the minimum number of characters required to make a given string to a palindrome if you are allowed to insert characters at any position of the string. For example, for the input "aab" the output should 1 (we'll add a 'b' in the beginning so it becomes "baab").

   The algorithm should run in $O(n^2)$ time if the input string has length n.

Idea: Use dynamic programming to create a n*n table to store the results of subproblems.

Define OPT$[a][z]$ = optimal maximum numer of insertions for converting the given string at index a and ending at index z. (1≤a≤z≤n)

Input: a given string with length n: `S`

Output: minimum number of characters required to make the input to a palindrome: `min_insection.`

Use dynamic programming - Memorization (Top Down).

Pseudocode:

```
plindrome(S):
  Let n = sizeof S
  Initialize OPT[n][n], a, z, gap     //a, z are index
  for gap in range(1, n):
    a = 0
    for z in range(gap, n):
      if S[a] == S[z]:
        OPT[a][z] = OPT[a+1][z-1]
      else                        //S[a] != S[z]
        OPT[a][z] = min(OPT[a][z-1], OPT[a+1][z]) + 1
      Endif
      a+=1
    Endfor
  Endfor
  min_insection = OPT[0][n - 1]
  return min_insection;
```

Time Complexity: $O(n^2)$, since table is $n * n$ size, the algorithm access each element once and each take O(1) time, so the total time is $O(n * n) = O(n^2)$

Correctness Proof: (by induction)

Base case: Suppose a string with n = 1 characters. Then the string itself is a palindrome.

IH: Assume a string with lenght n-1. Claim that the algorithm returns the minimum numner of character for the given string.

IS: Add a character to the left of the string, the length of the string now beomes n. We need to prove the IH works for string with length n.

Through the algorithm, the created table OPT $[a][z]$ store the minimum number of insertions needed to make the string[a....z] to a palindrome.

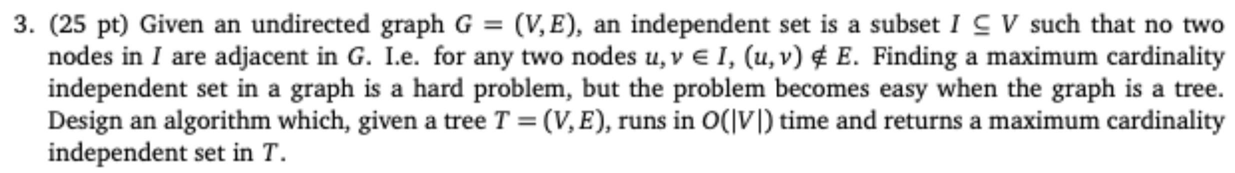
At each OPT$[a][z]$, there woud be two cases:

- S[a] == S[z]: It means there are two same characters in the symmertic postions of the string. In this case, no need to have inserted character. The algorithm reduced to subproblem OPT$[a+1][z+1]$
- S[a] ≠ S[z]: It means the characters in the symmertic positions are not identical. Need to have insertion.
    - Insert S[z] before S[a]: have one insertion S[z], algorithm reduces to subproblem OPT$[a][z-1]$
    - Insert S[a] after S[z]: have one insertion S[a], algorithm reduces to subproblem OPT$[a+1][z]$

The algorithm keep check the characters in the string from the both ends to middle, and will be done when it reaches the middle part (depends on if the length n is odd or even), which means S[a] == S[z-1]. There would be two cases happen:

- Single character: the character itself is palindrome, no need to have insertion.
- Two character: check these two characters
    - S[a] == S[z] : no need to have insertion.
    - S[a] == S[z] : Either insert S[a] after S[z] or insert S[z] before S[a].

Based on the above description, clearly the algorithm returns the correct answer. Proved.


3. (25 pt) Given an undirected graph $G = (V, E)$, an independent set is a subset $I \subseteq V$ such that no two nodes in I are adjacent in G. I.e. for any two nodes $u, v \in I$, $(u, v) \notin E$. Finding a maximum cardinality independent set in a graph is a hard problem, but the problem becomes easy when the graph is a tree. Design an algorithm which, given a tree $T = (V, E)$, runs in $O(|V|)$ time and returns a maximum cardinality independent set in T.

Input: a tree $T = (V, E)$

Output: A maximum cardinality independent set I

Algorithm:

```
Independent_set(T):
  Initialize I <- ∅,
  while(E is not empty):    // While loop keeps running until T has no edges
    Let v be a leaf node of minimum degree in T
    Let u be the parent node of v
    I <- I ∪ {v}          //Add node v to set I
    Remove nodes u and v from T
    Remove all the edges connecting to u and v from T  //Note that here will have
some                                                              // nodes with degree
0, these nodes                                                      //were used to be
connected with node u
  Endwhile
  let V be the set contains the remaining nodes which has 0 degree
  I <- I ∪ V    //add the remaining nodes which are also adjacent each other to set
I
  return I
```

Time Complexity: $O(|V|)$. Since the algorithm only visit each node of T once and each node takes O(1) time.

Correctness Proof (by induction):

Base case: Assume there are n = 1 node in tree T, which means it is the root. In this case, the root node itself is the maximum cardinality independset set I. As the algorithm explained, the root node could be considerd as leaf node, then it is added to set I and I is returned by the function. Clearly, the algorithm returns the correct answer.

IH: Claim that the algorithm returns maximum cardinality independset set I successfully for tree T with n-1 nodes size, assume a leaf node u in T.

IS: Add one more node v to node u, we need to prove the IH works for tree T with n nodes. Obviously the node v has been added is a leaf node, and node u is its parent.

According to the algorithm, node v woud be added to the I at the beginning. Then node v and its parent node u would be removed from tree T. The edges incident to them would be removed as well. Then we get a new tree T with some nodes which has degree 0 (they were used to be connected with node u). And the while loop keep running until there is no edges remaining in T (E is empty). There are only the remainning nodes with degree 0, clearly these nodes are adjacent each other and nodes in set I, then add these nodes in I. Now the set I is maximum cardinality independent.

Hence, the algorithm hold this case. Proved.

4. (10 pt) Consider the set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A (i.e., $B_i \subseteq A$ for each i). We say that a set $H \subseteq A$ is a hitting set for the collection $B_1, B_2, \ldots, B_m$ if H contains at least one element from each B_i—that is, if $H \cap B_i$ is not empty for each i (so H "hits" all the sets B_i).

We now define the Hitting Set Problem as follows. We are given a set $A = \{a_1, \ldots, a_n\}$, a collection $B_1, B_2, \ldots, B_m$ of subsets of A, and a number k. We are asked: Is there a hitting set $H \subseteq A$ for $B_1, \ldots, B_m$ so that the size of H is at most k?

Prove that the vertex cover problem $\leq_p$ the hitting set problem.

i.e., Polynomial-Time Reduction. Need to prove if the hitting set problem can be solved in polynomial time, then the vertex cover problem can also be solved in polynomial time. (a vertex cover of a graph is a set of vertices that includes at least one endpoint of every edge of the graph.)

Proof: Suppose the hitting set problem can be solved in polynomial time. Given an hitting set instance H for $G = (V, E)$ and of size k (positive integer). We construct the a vertex cover VC such that for each $e = (u, v) \in E$, let $B_e = \{u, v\}$, and of size k' = k.

Claim that G has a vertex cover of size at most k' iff the collection $\{B_1, B_2, \ldots, B_m\}$ has a hitting set H of size at most k = k'. The reduction function takes polynomial time $O(|E|)$.

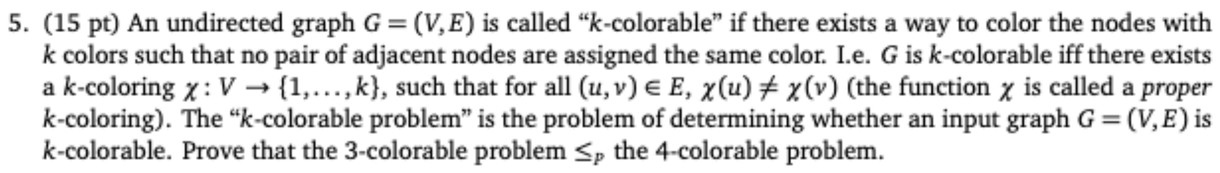
(=>): suppose G has a vertex cover VC of size at most k' = k. Then either $u \in VC$ or $v \in VC$ for all edge $e = (u, v)$. Then we can get that $\{u, v\} \cap VC \neq \emptyset$ for any set $\{u, v\}$ <=> $B_e \cap VC \neq \emptyset$ for any set B_e.

Hence, $H = VC$ is a hitting set of size k' = k.

(<=): Suppose H as a hitting of size at size k = k' for the collection of sets $\{B_e | e \in E\}$. Then $B_e \cap H \neq \emptyset$ for all $e \in E$ <=> $e(u, v) \cap H \neq \emptyset$. That is, each edge e is covered by an element if H.

Hence, H is a vertex cover for G of size k = k'.

Therefore, **the vertex cover problem $\leq_p$ the hitting set problem,** proved.



5. (15 pt) An undirected graph $G = (V, E)$ is called "k-colorable" if there exists a way to color the nodes with k colors such that no pair of adjacent nodes are assigned the same color. I.e. G is k-colorable iff there exists a k-coloring $\chi : V \to \{1, \ldots, k\}$, such that for all $(u, v) \in E$, $\chi(u) \neq \chi(v)$ (the function χ is called a *proper* k-coloring). The "k-colorable problem" is the problem of determining whether an input graph $G = (V, E)$ is k-colorable. Prove that the 3-colorable problem $\leq_p$ the 4-colorable problem.

Need to show if 4-colorable problem can be solved in polynomial time, then 3-colorable problem can also be solved in polynominal time.

Suppose the 4-colorable problem is solvable in polynomial time, then given graph $G = (V, E)$. The reduction function takes G as input and produces a new graph $G' = (V', E')$ such that: Let w be a vertex that is new (w is not in V). Then let $V' = V \bigcup \{w\}$ and $E' = E \bigcup \{\{v, w\} : v \in V\}$.

The reduction function takes ploynomial time $O(|V|)$.

(=>): Suppose $\chi : V \to \{1, 2, 3\}$, which is 3-coloring of G (G is 3-colorable). $\chi(u) \neq \chi(v)$ for all $(u, v) \in E$. Them define $\chi' : V' \to \{1, 2, 3, 4\}$ by $\chi'(v) = \chi(v)$ for $v \in V$ and $\chi'(w) = 4$.

Since there is no any vertiex v in G has color 4. For all $(u, v) \in E'$, $\chi'(u) \neq \chi'(v)$. So G' is 4-colorable .

Hence, if G is 3-colorable, then G' is 4-colorbale.

(<=): Suppose $\chi' : V' \to \{1, 2, 3, 4\}$, which is 4-coloring of G'. Without loss of generality, assign color 4 to w scuh that $\chi'(w) = 4$. We know that for all $(u, v) \in E'$, $\chi'(u) \neq \chi'(v)$.

Since w is connected to all vertices in V, so there are no vertex in v have $\chi'(v) = 4$. Remove w and its edges from G', then we can get a 3-coloring $\chi : V -> \{1, 2, 3\}$ defined by $\chi(v) = \chi'(v)$ for all $v \in V$.

Thus, if G' is 4-colorable then G is 3-colorbale.

Based on the abvoe describtion, **the 3-colorable problem $\leq_p$ the 4-colorable problem**. Proved.