Name(last, first): Chen, Joyce

ID (rightmost 4 digits): 5837

U C L A Computer Science Department

CS 180 Algorithms & Complexity

Final Exam **Total Time: 2:00 hours** **December 15, 2023**

*** **Write all algorithms in bullet form (as done in the past)** ***

You need to prove EVERY answer that you provide.

There are a total of 7 pages including this page.

You need to upload ONE file in PDF to Gradescope.

You can include at most 13 pages in your PDF.

Name(last, first): Chen, Joyce

1. **(15 points)** Consider an instance of the sequence alignment problem. Design an algorithm for solving it (you get full credit only for the most time efficient algorithm). Prove its correctness. Analyze its time complexity.

Algorithm:

crossings?

~~- run a modified version of mergesort on the sequence~~
~~- recursively divide sequence into 2 halves. keep a total count variable.~~
~~- when merging the 2 halves, do the following:~~
~~- if we are adding the element from the right half of the array, count += # of elements we haven't visited in left half yet.~~
~~- return total count, or the # of crossings there are in the sequence.~~

Longest common subsequence?

- create a 2D array opt, where opt [i] [j] is the longest common subsequence between string A ending on index i and string B ending on index j.
- opt [0] [0... N] = 0, opt [0... M] [0] = 0
- for i = 0 → len (A):

    for j = 0 → len (B):

        if A [i] == B [i]

            opt [i] [j] = opt [i-1] [j-1] + 1

        else:

            opt [i] [j] = max (opt [i-1] [j] , opt [i] [j-1])

- return opt [len (A)] [len (B)]

Proof:

- for the last character of the sequence A we're considering, we either add it to the longest common subsequence (1) or not (2).
- case (1):
  - we add it if there's a match between the last character in A and last character in B. thus, it would increase the existing subsequence we found for strings without last characters by 1.
- case (2):
  - we take max of whether the last character in LCS ended in A, or ended in B.
  - current characters of A, B are not added to LCS.

Time:

O (M*N), M = length of 1st string, N = length of 2nd string

Name(last, first): Chen, Joyce

2. **(20 points)** Find a subsequence of a given sequence such that the subsequence sum is as high as possible and the subsequence's elements are sorted in ascending order. This subsequence is not necessarily contiguous or unique. elements →

For example, consider subsequence {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11}. The maximum sum increasing subsequence is {8, 12, 14} which has sum 34.

opt(i,j) = opt(

Algorithm:

- let opt(i, j) = the max sum of an increasing subsequence ending on the jth element for a sequence of size i.

- create a 2D array "opt" that is of size N×N, N = # of elements in sequence.
- Set opt [0][0.....j] and opt [0.....i][0] = 0, and opt [1][0.....j] = the sequence given.
- for i = 0 ⇒ N:
    - create a global max variable
    - for j = 0 ⇒ i-1:
        - let local max = opt [i][j]
        - if local max > global max and sequence[i] > sequence[j]:
            - set global max to local max
    - opt [i][j] = global max + sequence[i]
      ↑ max ending at this element
- return opt [N][N]

Proof.

- Base case:
    - a sequence of size 0 will always have max sum = 0
    - a sequence of size 1 ~~end~~ ending at index n have max sum = max of all numbers in sequence from index 0 → n.
- Inductive step:
    - suppose we know the max sum for ending on nth element in sequence.
    - max sum for (n+1)th element would be the max sum possible for the previous n elements + the current (n+1)th element, assuming that the (n+1)th element is greater than the last element of the previous subsequence we're considering.
    - our algorithm checks for the max sum of all previous subsequences ending before n+1 and takes the max of them to add to current ending element we're checking.
    - It is correct

Time: we have 2 nested loops, each worst case N iterations. So total = $O(N^2)$.

Name(last, first): Chen, Joyce

3. **(20 points)** Given a string, count the number of times a given pattern appears in it as a subsequence.

string = "subsequence"
pattern = "sue"

Output: 7 for example: **subsequence** and **subsequence**

Algorithm.

idea:
- let opt(i,j) = times pattern ending on index j appears in the string ending on index i of string.
  if string[i] == pattern[j] (match):
    opt(i,j) = opt(i-1, j-1)
  if string[i] ≠ pattern[j]:
    opt(i,j) = opt(i-1, j)

Create a 2D array, opt where opt[i][j] = # times pattern ending on index j appears in the string ending on index i.

Initialize opt[0][0....n] = 0 and opt[0....n][0] = 0.

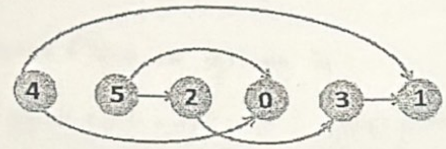
```
For i = 1 to len(string):
    For j = 1 to len(pattern):
        if string[i] == pattern[j] and j==1:
            opt[i][j] = opt[i-1][j-1]+1
        else if string[i] == pattern[j]:
            opt[i][j] = opt[i-1][j-1]
        else:
            opt[i][j] = opt[i-1][j]
Return opt[len(string)][len(pattern)]
```

Proof:
- Base case:
  - a pattern of size 0 will appear in string 0 times
  - a pattern of 1 letter will appear n times if the string contains n counts of that letter
- Inductive step:
  - suppose we find a matching between current last element of string, x, and some letter of the pattern, y.
  - then, the # of times the pattern from index 0 → index of y-1 appears in the string not containing the last element = times the entire pattern appears in entire string, because the last characters matched.
  - if no match, last character of string is not contributing to the count at all (it is not in the pattern), so we take the count of the pattern appearing in the string without the last character.

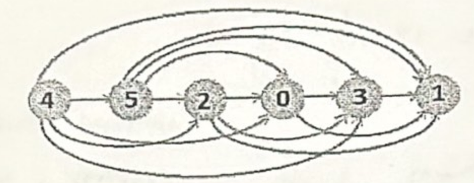
Time:
- creating nested for loops ⇒ outer goes from 1 – len(string), inner 1 – len(patt).
- total: O(M·N) where M = len of string and N = length of pattern.

Name(last, first): Chen, Joyce

4. **(15 points)** A DAG is given to us, we need to find maximum number of edges that can be added to this DAG, after which new graph still remain a DAG that means the reformed graph should have maximal number of edges, adding even single edge will create a cycle in graph



Input DAG



DAG with maximum edge addition

Algorithm:

- run topo sort on G
- consider each vertex v_i in G
  - add an edge going from v_j to v_i, given that v_j does not already have an edge pointing into v_i, and v_j comes before v_i.
- return G w/ the added edges

Proof:

- suppose that our algorithm doesn't find max edge addition DAG.
- then, adding an edge will not create a cycle.
- say we add an edge going from v-i to v-j.
  - all nodes that came before v-j have an outgoing edge to v-j already, based on our algorithm.
  - therefore v-i must come after v-j in the topological ordering.
  - based on algorithm, all nodes before v-i must point into v-i.
  - so v-j would have an edge going into v-i: $v_j \longrightarrow v_i$
  - adding edge (v-i, v-j) will produce a cycle: $v_j \rightleftarrows v_i$
  - >< contradiction!

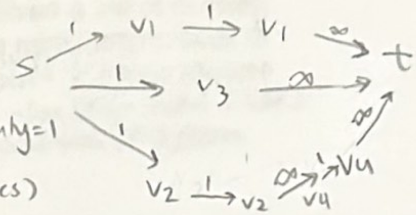
Time:

- topo sort: $O(V+E)$
- going through each $v\text{-}i \in V$: $O(V)$
- adding edges will take worst case $O(V)$ time.
- total: $O(V^2)$ time

Name(last, first): Chen, Joyce

5. **(15 points)** .Use max flow algorithm to solve the following problem. Given a graph with a source vertex s and a destination vertex t, find the minimum number of vertices which can be removed to make s and t disconnected.

Algorithm:



- run max flow on graph G:
  - split each node into 2 nodes with edge between capacity=1
  - each edge has capacity = ∞ (between 2 vertices)
  - Source → each vertex has capacity = 1
  - each edge going in to t has capacity = ∞
- the max flow = max # of paths going from s→t, given that there are no repeated vertices between 2 different paths.
- return total # vertices in G − max flow

Proof:

- each edge between 2 of the same vertices has 1 capacity, and ∞ capacity between 2 different vertices.
- this helps limit the # of paths going into a vertex to 1 path, because it has an out capacity of 1.
- capacity of edges connected to t have an ∞ capacity because that is the only node that can be repeated in the paths we find (given that it's the common end vertex of every path).
- therefore, we get max # of vertices we can traverse to get unique s−t paths by end of max-flow.
- min # of vertices to make s−t disconnected would thus = total vertices − max # that we need to keep s-t connected.

Time:

- modify graph to add edges and duplicate nodes — $O(E)$
- worst case there are N paths from s→t, so total = $O(E \cdot |f|)$, where $|f|$ = max flow.

Name(last, first): Chen, Joyce

6. (15 points) An airline company offers flights out of n airports on a daily basis. The flight time between any given pair of airports is known, but may differ on direction due to things like wind or geography (for example, Chicago to LA may take longer than LA to Chicago). All planes must leave at the scheduled time. Flights are identical on each day: starts at ~~7~~ 6 am and ends at midnight. Given a set of m daily flights that the airline company must schedule, determine the minimum number of planes that the company needs to purchase. **Example input -** 1: Boston (depart 6 A.M.) - Washington DC (arrive 7 A.M,). 2: Los Angeles (depart 7 A.M.) – San Diego (arrive 8 A.M.) 3: Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.). Can be done with TWO planes.

Algorithm:

- Create a network G:
  - each node represents a city and its corresponding time
  - add an edge from $v_1 \rightarrow v_2$ if there is a flight between those cities, with capacity = 1
  - add an edge from $v_1 \rightarrow v_2$ if they are the same city and time at v_2 is later than time at v_1, with capacity = ∞
  - source node s → each start of m flights have edge between w/ capacity = 1
  - for all nodes that are the last (not flying to any other city), connect to sink node t w/ edge capacity = ∞.
- Run max-flow on G.
- Return the max flow (min # of planes needed)

Proof:

- source node connected to n airports each w/ capacity 1 guarantees that we won't have more than m airplanes needed; worst case we need a plane for all m flights.
- edge cap. 1 between each flight ensures that only 1 airplane is allowed for that particular flight, no more no less. can't have more than 1 plane arriving at same destination from same source (assume that no flights are exactly overlapping!)
- edge cap ∞ between 2 same cities, because we can redirect a plane to start a new flight if it's already at the source location! don't need a new airplane for it.
- max flow = min # of planes, because that is the best way we can schedule flights such that a plane can fly again if already at source; and a new plane is needed for overlapping flight intervals.

Time:

- creating network takes $O(V+E)$ time
- max flow takes $O(|f| * E)$, where $|f|$ = max flow. total = $O(|f| * E)$.