Name: Michael Rogers

ID: 105667404

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**

Final Exam - 100 pts total **Fall 2019, CU-Boulder**

Notes:

- *Due date: 6pm on Sunday, December 8, 2019*
- *Submit a pdf file of your written answers to Gradescope and one py file with your Python codes to Canvas. All Python solutions should be clearly commented. Your codes need to run to get credit for your answers.*
- *You can ask clarification questions about the exam in office hours and on Piazza. However, please do not ask questions about how to answer a specific question. If there is confusion about any questions, we will address those issues at the beginning of class on December 5.*
- *All work on this exam needs to be independent. You may consult the textbook, the lecture notes and homework sets, but you should not use any other resources. If we suspect that you collaborated with anyone in the class or on the Internet, we will enforce the honor code policy strictly. If there is a reasonable doubt about an honor code violation, you will fail this course.*

---

1. (10 pts) Consider the following merge() algorithm to merge two sorted arrays into a larger sorted array. There are three errors in the algorithm.

```
MergeWithErrors(A, p, q, r)
    low = A[p..q]
    high = A[q..r]
    i = 0
    j = 0
    k = p
    while(i < q-p+1 and j < r-q)
        if(low[i] <= high[j])
            A[k] = low[i]
            j++
        else
            A[k] = high[j]
            i++
```

```
    k++
while(i < q-p+1)
    A[k] = low[i]
    i++
    k++
while(j < r-q)
    A[k] = high[j]
    j++
    k++
```

(a) (5 pts) List the three errors in the MergeWithErrors algorithm.

Solution. 1. The j++ in the first if satement should be i++ instead of j++.

2. The i ++ in the else statement should be j++ instead of i++.

3. In the first if statement, it should be: if(low[i] $<$ high[j]):

(b) (5 pts) For the following call to MergeWithErrors, what is the state of the array A after running MergeWithErrors. You can assume that the size of A won't change and values written outside the indices of A will be lost.

$$A = [0, 1, 3, 5, 2, 4, 6, 7]$$

$$MergeWithErrors(A, 0, 3, 7)$$

Solution. A = [0, 0, 0, 0, 0, 1, 3, 5]
Because we are incrementing j instead of i, the first element of low is going to be smaller than every element in high in this case. So the first 4 elements will be 0, and then when we finally increment i, we still take the first element of low which is zero. We then take the rest of low.

2. (25 pts) Let $G = (V, E)$ be a directed weighted graph of the pathways on the CU-Boulder campus, with edge weights being distances between different buildings/intersections. Engineering and Humanities are two vertices of G, and $k > 0$ is a given integer. Assume that you will stop at every building/intersection you pass by. A shortest k-stop path is a shortest path between two vertices with exactly k stops.

   (a) (5 pts) Provide an example showing that the shortest k-stop path can't necessarily be found using Breadth-first search or Dijkstras algorithm. You need an example for each algorithm that shows where it fails.

   Solution. Examples:
   BFS: The way BFS is structured, if a vertex is put into the queue that isn't optimal, then it will terminate without searching for a more optimal solution. BFS is strictly for finding a path from a to b. For example, if we were looking for a path from s-t and there were two paths from s-t, if the path with larger edge weight is queued first, then the algorithm will terminate before it has found the optimal solution.

   Dijkstra's: If our graph has a negative edge, Dijkstra's may not be able compute the shortest path from the source to the sink. For example, if every path going towards the destination was positive and all paths in line or backwards were considered negative, Dijkstra's may not find the optimal solution.

(b) (10 pts) Design an algorithm to find the shortest path from Engineering to Humanities that contains exactly k stops (excluding Engineering and Humanities). Notice that a k-stop path from these two buildings may not exist. So, your algorithm should also take care of such possibility. You need to provide an explanation of how your algorithm works to receive credit for this question.

Solution. We must use dynamic programming with tabulation to store sub-problems:

```
def dp-kPaths(G):
   m = create matrix of size k+1 x |V|
   set every element of m to infinity

   EC = 0
   for  i from 0 to k:
      for j equal to every node:
         if value of node + edge > j:
            m[i + 1][j] = value of node to edge

   if HUMN = INF:
      return "Not possible to reach distination"
   else:
      path = []
      backtrack from humanities storing each stop in path
      return path
```

(c) (10 pts) Implement your algorithm using the starter code provided on Canvas.

3. (25 pts) To entertain her kids during a recent snowstorm, Dr. Hoenigman invented a card game called EPIC!. In the two-player game, an even number of cards are laid out in a row, face up so that players can see the cards' values. On each card is written a positive integer, and players take turns removing a card from either end of the row and placing the card in their pile. The objective of the game is to collect the fewest points possible. The player whose cards add up to the lowest number after all cards have been selected wins the game.
One strategy is to use a greedy approach and simply pick the card at the end that is the smallest. However, this is not always optimal, as the following example shows: (The first player would win if she would first pick the 5 instead of the 4.)

4 2 6 5

(a) (10 pts) Write a non-greedy, efficient and optimal algorithm for a strategy to play EPIC!. The runtime needs to be less than or equal to $\theta(n^2)$. Player 1 will use this strategy and Player 2 will use a greedy strategy of choosing the smallest card. **Note: Your choice of algorithmic strategy really matters here. Think about the types of algorithms we've learned this semester when making your choice.** You need to provide an explanation of how your algorithm works to receive credit for this question.

Solution. Pseudocode:

```
def DPEpic(arr):
   dpTable = [[0 for x in range(len(arr)) for x in range(len(arr))

   for i in range(len(arr)):
      for j in range(i, len(arr)):
         if i + 2 <= j:
            select dpTable[i + 2][j]
         if i + 1 <= j - 1:
            select dpTable[i + 1][j - 1]
         if i <= j - 2:
            select dpTable[i][j - 1]

         dpTable[i][j] = min of the three options

   return dpTable[0][len(arr) - 1]
```

(b) (15 pts) Implement your strategy and the greedy strategy in Python and include code to simulate a game. Your simulation work for up to 100 cards, and values ranging from 1 to 100. Your simulation should include a randomly generated collection of cards and show the sum of cards in each hand at the end of the game.

4. (22 pts) In a previous homework assignment and classroom activity, we worked on the problem of finding the valley in an array, where array $A[1, 2, \cdots, n]$ with the property that the subarray $A[1..i]$ has the property that $A[j] > A[j+1]$ for $1 \leq j < i$, and the subarray $A[i..n]$ has the property that $A[j] < A[j+1]$ for $i \leq j < n$. For example, $A = [16, 15, 10, 9, 7, 3, 6, 8, 17, 23]$ is a valleyed array.

   Now consider the *multi-valley* generalization, in which the array contains k valleys, i.e., it contains k subarrays, each of which is itself a valleyed array. Suppose that $k = 2$ and we can guarantee that neither valley is closer than $n/4$ positions to the middle of the array, and that the "joining point" of the two singly-valleyed subarrays lays in the middle half of the array.

   (a) (8 pts) Now write an algorithm that returns the minimum element of A in sub-linear time.

   Solution. Pseudocode:

```
def findValley(subA, L, H):
   n = len(subA) - 1
   mid = L + (H - L) / 2
   if (mid == 0 or subA[mid - 1] >= subA[mid]) and (mid == n - 1 or
   subA[mid + 1] >= subA[mid]):
      return subA[mid]
   elif mid > 0 or subA[mid - 1] < subA[mid]:
      return findValley(subA, L, mid - 1)
   else:
      return findValley(subA, mid + 1, H)

def subLinFindValley(A):
   n = len(A) - 1
   low = A[0. . . n / 4]
   high = A[3n/4 . . . n]

   return min(findValley(low,0,len(low)-1), findValley(high,0,len(high)-1))
```

(b) (10 pts) Prove that your algorithm is correct. (Hint: prove that your algorithm's correctness follows from the correctness of another correct algorithm we already know.)

Solution. My algorithm contains a helper function called findValley that does the majority of the work for the algorithm.

subLinFindValley - This is the driver function of the program. Given that the two valleys are not closer than $\frac{n}{4}$ to the center point of the array, we can say that it is the case that there is not a valley from $\frac{n}{4}$ to $\frac{3n}{4}$.

$\therefore$ The middle half of the array can be ignored because our valley does not reside in the middle half.

findValley - Using induction:

Inductive Hypothesis: If findValley works to find a valley where a valley is definded as both the right and left neighbor are greater than the valley value for all inputs, then we can be sure that findValley will find the valley of an array.

Base Case: subA[mid - 1] >= subA[mid] and subA[mid + 1] >= subA[mid]. Since our array is valleyed, it means it is sorted up to the valley and after the valley. Assuming there aren't two valleys in the same sub list of A, then it must be the case that subA[mid] is the valley.

Inductive Step: Assuming there aren't two valleys on a single outer $\frac{1}{4}$ of the array, and the array is sorted in decending order up to the valley and sorted in ascending order after the valley, then the array is split into two equal length arrays similar to the strategy used in binary search. Since the array is sorted, every time it is split, if the element to the right of the valley is $<$ the valley, we can forget about the left side because the valley must be in the right half of the array as per the definition of a valley.

Conclution: Because it must be the case that the valley is in the smaller half of the array, and recursion stops when the mid = 0 or mid $<$ mid + 1 and mid -1 then our inductive hypothesis holds and the algorithm must be correct.

(c) (4 pts) Give a recurrence relation for its running time, and solve for its asymptotic behavior.

Solution. $T(n) = T(\frac{n}{2}) + \Theta(1)$

Using master's Theorem:

$log_b a = 0$, $k = 0$

Case 2: Since $log_b a = k$, then $T(n) \in \Theta(log_2 n)$

5. (18 pts) Suppose we are given a set of non-negative distinct numbers S and a target t. We want to find if there exists a subset of S that sums up to **exactly** t and the cardinality of this subset is k.
   Write a python program that takes an input array S, target t, and cardinality k, and returns the subset with cardinality k that adds to t if it exists, and returns $False$ otherwise. Your algorithm needs to run in $O(ntk)$ time where t is the target and n is the cardinality of S. In your code, provide a brief discussion of your runtime through comments, referring to specific elements of your code.

   For example -

   Input: s = {2,1,5,7}, t = 4, k = 2
   Output: False
   Explanation: No subset of size 2 sums to 4.

   Input: s = {2,1,5,7}, t = 6, k = 2
   Output: {1, 5}
   Explanation: Subset {1, 5} has size 2 and sums up to the target $t = 6$.

   Input: s = {2,1,5,7}, t = 6, k = 3
   Output: False
   Explanation: No subset of size 3 sums to 6.