Name: Alex Book
ID: 108073300

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.
Error 1: 'high' should be defined as 'A[q+1...r]', not 'A[q...r]'.
Error 2: In the 'if' statement in the first 'while' loop, 'j++' should be 'i++'.
Error 3: In the 'else' statement in the first 'while' loop, 'i++' should be '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.

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

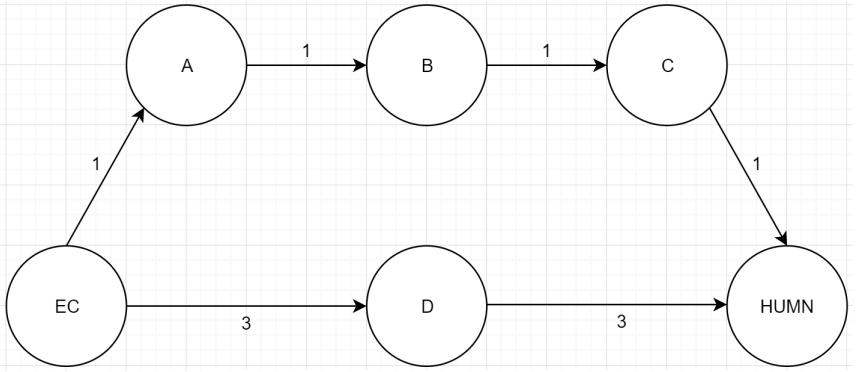
$$MergeWithErrors(A, 0, 3, 7)$$

Solution.
After that call, A=[0,0,0,0,0,1,3,5]. low[0]=0 is placed 4 times from comparing low[0] to high[0,1,2,3] during the first 'while' loop, then low[0,1,2,3]=[0,1,3,5] are placed into spots during the second 'while' loop.

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.



   Let k=2. Dijkstra's algorithm minimizes weight, but doesn't care about hitting a certain number of stops. It would find that the shortest path from EC to HUMN is EC-A-B-C-HUMN, having a weight of 4 and 4 stops. The EC-D-HUMN path has the desired number of stops, but a higher weight, so it isn't chosen.

   Let k=4. BFS minimizes the number of stops, but doesn't care about hitting a certain number of stops. It would find that the shortest path from EC to HUMN is EC-D-HUMN, having a weight of 6 and 2 edges. The EC-A-B-C-HUMN path has the desired number of stops, but a higher number of edges, so it isn't chosen.

(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.

My algorithm is a depth-first search function. I count all intermediate nodes and the destination nodes as "stops". In a call, there are four possibilities:

1. The current node is the destination and k=0. This means that a k-stop path from the source to the destination has been found. In this case, the destination node and a weight of zero are returned to the previous call (the handling of those two will be covered in a later case).

2. There is no k "budget" left, so the current path being analyzed is longer than k stops and is thus abandoned, returning the current node and a weight of infinity to the previous call.

3. The current node has no neighbors, thus abandoning the current path, returning the current node and a weight of infinity to the previous call.

4. The next node should be explored. In this case, we look at each of the current node's neighbors. For each one, we check if the weight of the edge from the current node to the next node added to the weight of the rest of the path down the line of recursion for that node is less than the current minimum path from the current node to the destination (if we have found a shorter viable path). If so, we reset the the current minimum path weight and append the vertices down that line of recursion to the 'ret' array. Once all neighbors have been analyzed, the current node is appended to 'ret'. Then 'ret' and the current minimum path weight are returned. For clarity, 'ret' holds the nodes that are part of the path from the current node to the destination if a proper path exists (meeting k-value requirements) and 'minSoFar' is the weight of that path. Each recursive call adjusts the next node to become the current node for that call, and subtracts 1 from k, as we "spend" one k-value each time we add another vertex to a path.

If there are no k-stop paths as desired, the initial call simply returns the source node and a weight of infinity. Otherwise, it returns the shortest k-stop path and its weight. The 'main' function then presents this data in a readable fashion.

(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 $\theta(n^2)$. 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.
   My solution centers around a recursive top-down with memoization approach. I use an n*n table, with cell [L][R] being the minimal sum for Player 1 considering the cards at indices L (the left card) and R (the right card). As soon as the function is called, I check to see if the optimal value for those cards exists yet (if it is non-infinity). If so, that value is returned in order to prevent unneeded further recursive calls. If not, my algorithm needs to fill that cell and thus considers all possible choices:
   Player 1 chooses the left card, then Player 2 chooses the minimum between the next left card and the right card.
   Player 1 chooses the right card, then Player 2 chooses the minimum between the next right card and the left card.
   The [L][R] cell is then set to the minimum sum between those two choices. To find that, it recurses all the way down through the "turns" to the base case, where two cards are next to each other, in which Player 1 chooses the minimum of those two cards and Player 2 gets the other of two cards. The minimum of those two choices is then returned by the function. Letting the "final" return (which is the initial call with L=0 and R=len(arr)-1) give the minimum possible sum for Player

1. This score is subtracted from the total sum of the array in order to give the score for Player 2.

(b) (15 pts) Implement your strategy and the greedy strategy in Python and include code to simulate a game. Your simulation should 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 peak 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 valley-ed 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 valley-ed 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-valley-ed 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.

```
def valleys(arr):
    quarter=(len(arr))//4
    ret=[]
    ret.append(findValleys(arr,0,quarter-1))
    ret.append(findValleys(arr,3*quarter,len(arr)-1))
    if len(ret)>0:
        return min(ret)
    else:
        return # no valley

def findValleys(arr,p,r):
    if p<=r:
        mid=p+(r-p)//2
        if arr[mid]<arr[mid-1] and arr[mid]<arr[mid+1]:
            return arr[mid] # valley found
        elif arr[mid]>arr[mid-1]:
            return findValleys(arr,p,mid)
        else:
            return findValleys(arr,mid,r)
    else:
        return
```

(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.
The algorithm used mainly utilizes the findValleys function, so we must consider that function's correctness. I'm unsure as to whether or not a loop invariant proof would be accepted for the recursive calls to 'findValleys', so beneath this proof is a straightforward logical argument explaining the algorithm (scroll to next page).

Loop Invariant Proof
Loop invariant:
Before each function call, the valley of the subarray is somewhere in arr[p...r].

Initialization:
Before the first function call, the function hasn't yet done anything with the passed in array and indices, so the valley must be in the given subarray.

Maintenance:
There are 3 possible cases in the function itself.
Case 1: The 'mid' value is the valley. In this case, the valley has been found, and the loop exits.
Case 2: The valley is to the left of 'mid' value. In this case, the 'r' value is set to mid because we can ignore everything right of 'mid'. So at the beginning of the next call, the valley will be between p and r.
Case 3: The valley is to the right of 'mid' value. In this case, the 'l' value is set to mid because we can ignore everything left of 'mid'. So at the beginning of the next call, the valley will be between p and r.

Termination:
The function terminates when the 'mid' value is the valley. The 'mid' value is returned to be appended to the list of valleys 'ret' in the 'valleys' function.

Logical Argument
There are 3 possible cases:
Case 1: The 'mid' value is the valley. In this case, the valley has been found, and the function returns said valley.

Case 2: The valley is to the left of 'mid' value. In this case, the 'r' value is set to mid because we can ignore everything right of 'mid'. So in the next call, the valley will be between p and r.

Case 3: The valley is to the right of 'mid' value. In this case, the 'l' value is set to mid because we can ignore everything left of 'mid'. So in the next call, the valley will be between p and r.

Since the given subarray must be valley-ed (as described in the problem), Case 1 will eventually be achieved, and the function returns the valley.

Conclusion For Both LI Proof and Logical Argument
So, since 'findValleys' correctly returns the valley in a given subarray, we need only to look at 'valleys'. This function takes the two found valleys and returns the minimum value of the two, which must be the minimum value in the entire given array. Therefore, the algorithm functions correctly.

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

Solution.
Let T(n) represent the runtime of the entire algorithm (the 'valleys' function) and V(n) represent the runtime of the 'findValleys' function.

$T(n) = 2V(\frac{n}{4}) + \Theta(1)$
$V(n) = V(\frac{n}{2}) + \Theta(1)$

Runtime of $V(n)$ by Master's Theorem $= O(logn)$
Substitute into $T(n)$:
$T(n) = 2 * O(log(\frac{n}{4})) + \Theta(1)$
$T(n) = 2 * (O(logn) - O(log4)) + \Theta(1)$ (using logarithm rules)
$T(n) = 2 * O(logn) - 2 * O(log4) + \Theta(1)$
$T(n) = O(logn)$

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 $O(nt)$ 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.