# Sophomore Year CS-2500 Algorithm’s Honor’s Project --- Kattis Programming Problem Survey Over Algorithm Technique, Complexity Class, and Correctness

## Introduction

The objective of this project is to challenge myself and use what I’ve learned in my Algorithm’s course to solve Kattis problems ranging from easy to hard. To do this we will be solving 10 problems using the Python programming language. For each problem we will find a solution, explain the algorithm technique(s) used to solve, the time complexity, and an argument for the correctness of the problem solution. Every problem was chosen by rolling two dice, a d6 and d10. The d6 was for the tens place and acted as a 0 for 1-2, 1 for 3-4, and a 2 for 5-6. A d10 was obviously used for the digits 0-9, a 10 acting as the 0. This paper simply contains the solution for each Kattis problem I solved, along with all of the other information previously discussed.

## Kattis Algorithms

### Branch Manager

import sys

sys.setrecursionlimit(2 \* 10\*\*5)

input = sys.stdin.readline

num\_cities, num\_people = map(int, input().split())

adjacency\_list = [[] for \_ in range(num\_cities)]

parent = [0] \* num\_cities

for \_ in range(num\_cities - 1):

a, b = map(int, input().split())

a -= 1

b -= 1

parent[b] = a

adjacency\_list[a].append(b)

for children in adjacency\_list:

children.sort()

entry\_time = [-1] \* num\_cities

exit\_time = [-1] \* num\_cities

time = 0

def dfs(node):

global time

entry\_time[node] = time

time += 1

for child in adjacency\_list[node]:

dfs(child)

exit\_time[node] = time

dfs(0)

max\_entry\_seen = -1

for i in range(num\_people):

destination = int(input()) - 1

if exit\_time[destination] <= max\_entry\_seen:

print(i)

exit(0)

max\_entry\_seen = max(max\_entry\_seen, entry\_time[destination])

print(num\_people)

For this algorithm the major algorithmic technique we use is Transform-and-conquer. Here we are taking the orginal problem and transforming it into a form where comparison are done using entry and exit times with DFS. The time complexity for this algorithm simplifies to O(nlogn) or linear logarithmic because of the sorting algorithm used for sorting the children in the adjaceny list. We can prove correctness by first looking at the inductive dfs divide and conquer algorithm used to set entry and exit times for each child in the adjacency list. Our base case is that the root node, 0 , is set to time = 0 before any recursive calls are made. From there we can assume that for any node n we correctly set the entry\_time[n] and exit\_time[n]. When we call the DFS on a node, it will recursively call DFS on all children n. With our hypthoses we made we can correctly assign an entry and exit time to each of the n children or to each of the node’s descendants. Now for the processing of people and their destinations we can make an argument using loop invariant. At the start of each iteration, max\_entry\_seen will store the largest entry time seen and is initialized to -1. If the exit\_time[destination] <= max\_entry\_seen we will print the number of people passed and exit because the current destination attempting to be reached has already been cut off by someone else having crossed the path. This problem was by far the hardest out of all of them and took me the longest to figure out, I had to outsource in order to try and find a way to be able to finnish this problem in the time it wanted me to. My original answer was somehwere around O(n^2) time complexity, which was not good enough for what I had to turn in.

### Class Field Trip

string1 = input()

string2 = input()

string3 = ''.join(sorted(string1+string2))

print(string3)

For this next algorithm the best way to describe the technique used here is just a basic sorting algorithm. The sorting itself is done with a built in python fuction “sorted” which takes O(nlogn) time. Correctness can be proved with the iterative approach of python’s sorted function. After sorted is complete we will have a string which is sorted in ascending order.

### Creative Accounting

days, minSegSize, maxSegSize = map(int, input().split())

prefixSum = [0] \* (days+1)

dailyProfits = [0] \* (days)

sum = 0

for i in range(days):

num = int(input())

sum += num

prefixSum[i+1] = sum

dailyProfits[i] = num

minProfitSegs, maxProfitSegs = float("inf"), float("-inf")

for segSize in range(minSegSize, maxSegSize + 1):

for offset in range(segSize):

start = offset

profitCount = 0

#print(f"TESTING: SegSize: {segSize} start: {start}") TESTING

if(offset > 0):

end = offset

segSum = prefixSum[end]

'''

###FOR TESTING###

print('< ', end="")

for num in range(end):

print(f'{dailyProfits[num]}, ', end="")

print('>')

'''

if (segSum > 0):

profitCount += 1

while (start < days):

end = min(days, start+segSize)

segSum = prefixSum[end] - prefixSum[start]

'''

###FOR TESTING###

print('< ', end="")

for num in range(start, end):

print(f'{dailyProfits[num]}, ', end="")

print('>')

'''

if (segSum > 0):

profitCount += 1

start += segSize

#print(f'PROFIT: {profitCount}') TESTING

minProfitSegs = min(minProfitSegs, profitCount)

maxProfitSegs = max(maxProfitSegs, profitCount)

print(f"{minProfitSegs} {maxProfitSegs}")

This is most closely related to a brute force algorithmic approach, this is because we are iterating through every possible permuation of segment sizes in order to find both the minimum and maximum profit. This also however comes at the cost that in the worst case scenario this problem because O(n^2) or quadratic as we are looping within a loop. Since this is an iterative algorithm we can prove its correctness with loop invariant. At the start of each iteration we know that every segSize at each offset have count profit sums and prefix sums. To maintain this we go through each segment and offset and compute the prefixSum[end] – prefixSum[start], updating the amount of profitable segments when this number is greater than 0. Finally we update the minimum and maximum profitable segments before entering the next iteration.

### d.) Fading Wind

import math

startHeight, fixedFactor, startingVelocity, windStrength = map(int, input().split())

distance = 0

while startHeight > 0:

startingVelocity += windStrength

startingVelocity -= max(1, math.floor(startingVelocity/10))

if startingVelocity >= fixedFactor:

startHeight += 1

elif startingVelocity > 0 and startingVelocity < fixedFactor:

startHeight -= 1

if startHeight == 0:

startingVelocity = 0

if startingVelocity <= 0:

startHeight = 0

startingVelocity = 0

if windStrength > 0:

windStrength -= 1

distance += startingVelocity

print(distance)

The algorithmic technique used in this program is just brute force, we iterate through until we reach the ground. The time complexity for this program is O(n) linear time. This is an iterative program so we can prove the correctness via loop invariancy. At the beginning of each iteration we have are set startHeight, velocity, fixedFactor, and windStrength that we will edit during the algorithm. We also accumalte distance over time as a variable that changes depending on our velocity. The algorithm ends when our height reaches zero on in other words we reach the ground.

### Ghost Leg

elements, numRungs = map(int, input().split())

rungList = [int(input()) for x in range(numRungs)]

permList = [x+1 for x in range(elements)]

for i in range(len(rungList)):

numIndex1 = rungList[i]-1

numIndex2 = rungList[i]

temp = permList[numIndex1]

permList[numIndex1] = permList[numIndex2]

permList[numIndex2] = temp

for num in permList:

print(num)

The algorithmic technique used for this program is actually a transform and conquer type. Instead of following the rungs down to find where each is in the new array, we can just run through the entire list one time and use a swapping algorithm to get the same results. The time complexity for this program is O(n) linear as it is one for loop. We can prove its correctness with loop invariancy because it is an iterative program. Before each iteration we have a rungList that holds the positions of the rungs and a permList which is a copy of the original list but will be used for swapping elements. The algorithm will go through each rung in our rungList and swap the elements of the permList according to what rung they are a part of, terminating when we reach the end of our rungList.

### MazeMan

from collections import deque

import heapq

import copy

mazePrintedCount = 1

def PrintMaze(m):

global mazePrintedCount

print(f'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MAZE:{mazePrintedCount}')

for r in range(rows):

for c in range(columns):

print(m[r][c],end="")

print()

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

mazePrintedCount+=1

rows, columns = map(int, input().split())

maze = []

totalDots = 0

for \_ in range(rows):

line = list(input())

totalDots += line.count(".")

maze.append(line)

entrancePosition = []

for c in range(columns):

if 'A' <= maze[0][c] <= 'W':

entrancePosition.append((0, c))

if 'A' <= maze[rows - 1][c] <= 'W':

entrancePosition.append((rows - 1, c))

for r in range(1, rows - 1):

if 'A' <= maze[r][0] <= 'W':

entrancePosition.append((r, 0))

if 'A' <= maze[r][columns - 1] <= 'W':

entrancePosition.append((r, columns - 1))

#PrintMaze(maze)

#tempMaze = maze #copy so wont mess up original [THIS IS FOR TESTING]

moveDirections = [[-1,0],[1,0],[0,-1],[0,1]] #up, down, left, right

reachablesDots = {} #key = entrance; value = dot coordinate

visited = {} #universal visited for decreasing amount of mazes generated

for entrance in entrancePosition:

#tempMaze = copy.deepcopy(maze)

#print(f'$$$$$$$$$$$$$$$$$ FOR ENTRANCE {maze[entrance[0]][entrance[1]]} $$$$$$$$$$$$$$$$$')

entranceChar = maze[entrance[0]][entrance[1]]

if entranceChar not in reachablesDots:

reachablesDots[entranceChar] = set()

queue = deque([entrance])

visited[entrance] = entranceChar

while queue:

pos = queue.pop()

#tempMaze[pos[0]][pos[1]] = "\*"

#PrintMaze(tempMaze)

for dx, dy in moveDirections:

nx, ny = [pos[0]+dx,pos[1]+dy]

if not (0 <= nx < rows and 0 <= ny < columns):

continue

if maze[nx][ny] not in {".", " "}:

continue

if (nx, ny) in visited:

otherEntrance = visited[(nx, ny)]

if otherEntrance != entranceChar:

#print(f'GOT OTHER ENTRANCE {entrance} FOUND {otherEntrance}')

reachablesDots[entranceChar].update(reachablesDots[otherEntrance])

for dot in reachablesDots[otherEntrance]:

visited[dot] = entranceChar

continue

else:

visited[(nx, ny)] = entranceChar

if maze[nx][ny] == ".":

reachablesDots[entranceChar].add((nx, ny))

queue.append((nx, ny))

#print("$$$$$$$$$$$$$$$$$")

#print(reachablesDots)

allDots = set()

for dots in reachablesDots.values():

allDots.update(dots)

selectedEntrances = []

dotsCovered = set()

heap = []

for entrance in reachablesDots:

heapq.heappush(heap, (-len(reachablesDots[entrance]), entrance))

while dotsCovered != allDots and heap:

\_, bestEntrance = heapq.heappop(heap)

newDots = reachablesDots[bestEntrance] - dotsCovered

if not newDots:

continue

selectedEntrances.append(bestEntrance)

dotsCovered.update(newDots)

#print(selectedEntrances)

#print(dotsCovered)

print(len(selectedEntrances), totalDots - len(allDots))

The main algorithmic techniques used in this program is divide and conquer with a BFS algorithm for graph traversal. What’s interesting about this program is that we actually use BFS smart here, because we need to BFS all of the entracnes to find every possible dot that can be found. However, if we have a universal visited set we can easily traverse across all entrances and when any entrance visits an already visited dot. We can just give that entrance the same set of dots as the entrance that found it first, because any entrances connected to each because of BFS algorithm will have the same set of dots. So we don’t need to BFS for every entrance necessarily, we just need to do it for one big one and the rest might fall in line easily. This is the main idea of this program that got me through the work of making it past the required time limit. Besides this we take advantage of sets and heaps to both get the total amount of entrances needed to get all of the dots that are findable and to find how many dots are left untouched. We do this by pushing each entrance’s set of found dots to essentially a max heap by negating their amount of found dots along with the entrance it belongs to as a tuple. With this we can use a while loop to go through until we’ve gone over every dot and through the entire heap. Here we take the max of the heap (the entrance with the largest amount of dots) and use a function of sets in python by taking the difference between the taken heap dot set and subtracting the dots covered set. Essentially giving us how many dots have not been covered. IF NO NEW DOTS ARE COVERED we can continue, as we don’t need this entrance. Otherwise we add the entrance to a list of ones we need and update the dots we have discovered. The time complexity of this program comes from our worst case which is our Max-Heap which is linear logarithmic worst case or O(nlogn). We can prove the correctness of this program by first looking at our BFS which would be iterative in nature, so we can use loop invariancy. At the beginning of each iteration we have a queue filled with unvisited nodes, we have moveDirections that the BFS will use to traverse the graph, a reachableDots dictionary to keep track of what dots are reachable by what entrance, and a universal visited set to keep track of what dots have been visited by ALL entrances. During the iteration we go through the queue, popping the first element, and from the position of the entrances traverse the graph by iterating through moveDirections. If we are out of bounds or don’t find an empty space or “.” we continue. If we do we find out if the position has been already visited, if so we can easily update the set of found dots from our current entrance with the found entrance’s dots and end the BFS for that entrance. Otherwise we traverse the graph as normal and mark the dot as visited along with the entrance that it was visited by and add the the dot to our reachableDots dictionary for later use during our Max-Heap algorithm. At the end of this BFS algorithm we will successfully have made our reachableDots dictionary which stores all of the entrances along with all of the dots that they can reach. From here we can prove the correctness of our Max-Heap program in the same way using loop invariancy. At the beginning of each iteration we have a selectedEntrances list that stores all of the entrances that are needed to find all of the findable dots, we have a dotsCovered set that will store how many dots we have iterated through during our algorithm’s life, and a heap that acts as a Max-Heap that stores the entrances by the largest amount of dots they cover. During the iteration of our program we first pop the first element from our heap, giving us the entrance with the largest amount of covered dots. We use this to find how many newDots we have covered during this algorithm and then add the entrance if it is necessary in finding new dots. By the end of this program we have the number of entrances that are necessary to find all of the dots possibly findable.

### Metronome

songLength = float(input())

print(songLength/4.0)

Their isn’t really an algorithmic technique for this program as it’s two lines. The complexity time for this program is O(1) constant. I mean there is not much to say about this one, you take in an input, you give an output.

### Platform Placing

import sys

numFoundationPoints, smallestPlatformLength, largestPlatformLength = map(int, input().split())

foundationPoints = [int(input()) for x in range(numFoundationPoints)]

foundationPoints = sorted(foundationPoints)

lengths = [largestPlatformLength] \* numFoundationPoints

for i in range(numFoundationPoints-1):

if (lengths[i] + lengths[i+1] > 2 \* (foundationPoints[i+1] - foundationPoints[i])):

newLength1 = min(lengths[i], 2 \* (foundationPoints[i+1] - foundationPoints[i]) - smallestPlatformLength)

newLength2 = 2 \* (foundationPoints[i+1] - foundationPoints[i]) - newLength1

if (newLength1 < smallestPlatformLength):

print(-1)

sys.exit()

lengths[i] = newLength1

lengths[i+1] = newLength2

#print(f'\*\*\*{lengths}\*\*\*')

print(sum(lengths))

The algorithmic technique used for this program is actually a mathematical greedy algorithm. The biggest mind blowing solution to this problem is some math we can derive. We know that for every platform in ascending order *xi* that *xi + li/2* or the position plus its rightward length MUST be less then or equal to *xi+1 – li2/2* or the next platform’s leftward length. With this we can dervive an equation that gives us all of the position on one side and the lengths on the other, *li + li2 <= 2(xi+1 - xi)*. With this, the program starts to come together and we can iterate through a sorted list of the foundation points, as the algorithm would obviously break if we tried to do this with random unsorted platforms. This way we only have to pay attention to the rightward direction and adjust the leftward if needed. The time complexity of this algorithm is in worst case scenario O(nlogn) or linear logarithmic because of the built in python sorted function to rearrange the foundationPoints in ascending order. We can prove the correctness of this algorithm by using loop invariancy since it is an iterative program. At the beginning of each iteration we have a foundationPoints list that holds the sorted order of the position of the foundation points, we also have a list of lengths that was greedily initialized to the maximum allowed platform length for every platform. During an iteration of the program we go through each foundation and we use the formula we derived to check to see if the platform currently fits in the space or not. If it does we continue to the next foundation, otherwise we will give a new length to both the platform we are on and the next platform we are overlapped with currently. We do this by using some formulas. For newLength1 we choose the minimum of either the current length of the platform or twice the gap (the right side of the derived formula) – the smallest platform length available. We then do the same thing for newLength2 by multiplying the gap by 2 and subtracting our first length to dynamically change the second platform to what it needs to fit the first. We will then check to see if the newLength1 we just calculated is less then what we can accept as the smallest platform, which would make this impossible to complete. Otherwise we assign the new lengths to our current platform and the next. At the end of this algorithm we will have our length’s list that has the maximum possible lengths for each platform possible to make up the most amount of space possible.

### Room Evacuation

from collections import defaultdict

def printFloorPlan(plan: list) -> None:

print("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*FLOOR PLAN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",end="\n")

for x in range(len(plan)):

for char in plan[x]:

print(char,end=" ")

print()

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")

def bpm(u, visited):

for v in graph[u]:

if not visited[v]:

visited[v] = True

if match\_to[v] == -1 or bpm(match\_to[v], visited):

match\_to[v] = u

return True

return False

height, width, time = map(int, input().split())

exitPositions = []

playerPositions = []

distances = [-1]\*height

floorPlan = [0]\*height

for h in range(height):

newline = input()

col = [0]\*width

dist = [-1]\*width

for w in range(width):

col[w] = (newline[w])

if (newline[w] == "E"):

exitPositions.append([h,w])

elif (newline[w] == "P"):

playerPositions.append([h,w])

floorPlan[h] = col

distances[h] = dist

ct = 0

myQueue = []

for exit in exitPositions:

distances[exit[0]][exit[1]] = 0

myQueue.append(exit)

directions = [[-1,0],[1,0],[0,-1],[0,1]]

while len(myQueue) > 0:

#printFloorPlan(distances) #TESTING PURPOSES

ct += 1

point = myQueue.pop(0)

for dir in directions:

newX = point[0] + dir[0]

newY = point[1] + dir[1]

if (newX >= 0 and newX < height and newY >= 0 and newY < width and distances[newX][newY] == -1 and floorPlan[newX][newY] != "#"):

distances[newX][newY] = distances[point[0]][point[1]] + 1

myQueue.append([newX,newY])

graph = defaultdict(list)

exitTimeDictionary = {}

exitID = 0

for pid, (px, py) in enumerate(playerPositions):

for t in range(time + 1):

for ex, ey in exitPositions:

if distances[px][py] == -1:

continue

if distances[px][py] <= t:

key = (ex, ey, t)

if key not in exitTimeDictionary:

exitTimeDictionary[key] = exitID

exitID += 1

graph[pid].append(exitTimeDictionary[key])

V = exitID

U = len(playerPositions)

match\_to = [-1] \* V

result = 0

for u in range(U):

visited = [False] \* V

if bpm(u, visited):

result += 1

print(result)

A couple of algorithmic techniques were used here including BFS, Biparte Graph, and DFS. This was either the 1st or 2nd hardest problem here. The main idea of this problem was to find the closest escape route for a person to take within the time constraints, with the extra constraint that no two people could be on the same space at the same time. The latter constraint is very important and what made this problem so difficult, hence the biparte graph. The main idea is that we first BFS from every exit to get the distance of each point from the nearest exit. Now where it gets difficult is when we have to implement the time constraint, because what this means is that no two people can be using the same exit at the same time. Hence we first make a biparte graph to match each specific player to a set of possible exits at different times. This is useful because only ONE player can occupy an exit at a certain time. Hence we then use a DFS algorithm to match each player to a specific exit at a specific time in the biparte graph. I believe in worst case this algorithm can be O(n^3) time complexity, this Is because of our three nested loops building the biparte graph. We can prove the correctness of this algorithm using loop invariancy and logical deduction, since the program works in distinct stages that build toward the final result. At the beginning, we read the dimensions and create a 2D list that stores the layout of the floor plan. Each point is then scanned to collect the positions of all players and all exits. We then initialize a distance matrix that helps us keep track of how many steps it takes to reach each square from any exit. We do this by performing a multi-source BFS from all exit points, allowing us to know exactly how far each player is from every exit they could potentially reach. The next step involves preparing the bipartite graph. We assign each player a node on the left side of the bipartite graph, and each (exit, time) pair a node on the right side. The reason we include time is because only one player can use an exit at any one time unit. This allows us to convert the problem into a maximum bipartite matching problem, where each player must be matched to a unique exit-time slot. For every player, we loop through all exits and all possible time steps from 0 to the max allowed time, and if the distance from the player to the exit is less than or equal to the time step, we create an edge between that player and that (exit, time) node. After building the graph, we then try to match as many players as possible using a standard DFS-based approach for bipartite matching. This works because we can repeatedly try to find an augmenting path for each unmatched player. If we do, we assign that player a spot and mark it as matched. The match is only successful if it doesn't overlap with an already-matched player unless an alternating path can free up a spot. We keep track of all matches and return the total number of successful ones at the end. This algorithm works correctly because it builds up the graph only with valid movements (distance ≤ time), ensures each player gets only one exit, and prevents two players from using the same exit at the same time. The BFS guarantees shortest paths, and the matching ensures maximum pairing under the constraints. So by the time the program finishes, we’ve found the maximum number of players that can escape within the allowed time limit.

### Streets Ahead

streets, drivers = map(int, input().split())

streetNames = [input() for x in range(streets)]

streetIndices = {name: i for i,name in enumerate(streetNames)}

driverDirections = [input().split() for i in range(drivers)]

for driver in driverDirections:

start = driver[0]

end = driver[1]

print(abs(streetIndices.get(end) - streetIndices.get(start))-1)

The main algorithmic technique used for this problem is mostly hashing for fast lookup times. This is because the main idea of this program is to find how many streets each car crosses while driving, we can process the streets into a dictionary for fast lookup later. The time complexity of this program is linear because of the for loop and just general building of the dictionaries. We can prove the correctness of this program with loop invariancy as it is an iterative program. Before each iteration we have list of driverDirections that holds the streets in which each driver begins and ends. We also have a dictionary called streetIndices that holds the streets and their positions. During an iteration of this program we will iterate over each driver’s beginning street and ending street. We can use this information to get the position of the stree the driver is starting at and the position of the stree the driver is stopping at by using our streetIndices dictionary. We subtract the end and the start making sure to use absolute value (To be sure we take into account if the driver is going in a negative or positive direction) and subtract by 1 to not include the street we end at.

## 3.) Conclusion

The goal of this project was to use the lesson’s I learned in my Algorithm’s course to solve real problems. Throughough my time working on these problems, I feel as though I’ve gotten a better grasp on my algorithmic thinking. In other words, I’ve enhanced my problem solving skills and improved the way I approach tough problems. Some of these problems were a real challenge, took me a long time, and I was forced to outsource for solutions that I just couldn’t figure out on my own. In the future this experience will come back to benefit me, as the trials I’ve endured working on this project have made me better than I was before. Overall, this project has contributed greatly to my growth as a programmer as well as making me a better fit for my future career.