Author: Liam Coleman and Aidan Dalgarno-Platt  
  
Problem 1)

The algorithm first takes in the input and makes a list of edges, which takes O(m) time. It then creates a graph in the form of a list using the edges list. Each index represents a location, numbered 0…n-1, and the value at each index is a list in the form [neighboring location, cost, time]. Creating the initial graph takes O(n) time, and adding all the edges takes O(m) time.

Next, the algorithm creates a “visited” set and a priority queue of tuples representing edges in the graph in the form (neighboring location, cost, time) in order to perform an adjusted version of Dijkstra’s Algorithm. Dijkstra’s algorithm continues as usual except that every iteration, it checks if the popped location is the last node and if the accumulated cost and time are within the monetary budget and time budget. If so, it outputs “YES”. If not, it continues, and if every location is visited, it outputs “NO”. Our implementation of Dijkstra’s Algorithm uses heapq to make the priority queue and takes O(mlog(n)) time.

Dijkstra’s Algorithm always finds the shortest path for a weighted graph like this one. Our algorithm just keeps track of the monetary and time costs, two different weights instead of one. In total, the program takes O(m + n + m + mlog(n)) time, which is O(mlog(n)).

Problem 2)



Iteration 1



Iteration 2



Iteration 3



In this step, the flow is pushed back from A->T and through A->C and then C->T

Problem 3)

First, the algorithm creates a list of student courses, which takes O(num\_students) time, and a list of course limits, which takes O(num\_courses) time. It then makes a call to max\_registered() and starts processing. Populating course\_interests takes O(num\_students \* num\_courses).

Next, the courses are sorted in descending order by course limits so that the courses with the most open slots are checked first, which takes O(num\_courses \* log(num\_courses)). Then an assigned\_courses list is created with a spot for each student to keep track of which courses they’ve enrolled in, which takes O(num\_students) time.

Finally, the algorithm iterates through each course, and for each course iterates through every student willing and interested to take it. Then, if that student hasn’t enrolled in three courses already and that course has slots remaining, they enroll, the number of total registrations increases by 1, and the number of spots remaining in that course decreases by 1. This all takes O(num\_students \* num\_courses) time.

Populating the courses with the most slots available just works, since as many students who were interested in that course will get in that way, and the courses with the fewest available slots won’t be taken up by students who could have settled for another course with more slots available. In all, the algorithm takes O(num\_students + num\_courses + num\_students \* num\_courses + num\_courses \* log(num\_courses) + num\_students + num\_students \* num\_courses) time, which is O(num\_students \* num\_courses) overall.