```
import sys

def source_to_transit_capacity(sources,transit,destination):
    capacity_string = ""
    #generates the arguments that cik for each source transit pair <= link capacity
    for i in range(1,sources +1):
        for k in range(1, transit + 1):
            entry = ""
            for j in range(1,destination+1):
                entry += "x{}{}{} + ".format(i,k,j)
            entry = entry[0:-2]
            entry += "- c{}{} <= 0 \n".format(i,k)
            capacity_string += entry
    return capacity_string

def transit_to_desination_capacity(sources,transit,destination):
    capacity_string = ""
    for j in range(1,destination+1):
        for k in range(1, transit + 1):
            entry = ""
            for i in range(1, sources + 1):
                entry += "x{}{}{} + ".format(i,k,j)
            entry = entry[0:-2]
            entry += "- d{}{} <= 0 \n".format(k,j)
            capacity_string += entry
    return capacity_string

def source_to_dest_demand_volume(sources,transit,destination):
    #generates the load count hij = i + j for each path x ikj
    demand_string = ""
    # in sources, in destination (create one for each in next), in transit
    for i in range(1,sources + 1):
        for j in range(1,destination + 1):
            #source * destination entires, create here
            entry = ""
            for k in range(1,transit+1):
                entry += ("x{}{}{} + ".format(i,k,j))
            entry = entry[0:-2]
            entry += "= {} \n".format(i+j)
            demand_string += entry
    return demand_string

def split_along_two_paths(sources,transit,destination):
    binary_string = ""
    #generates and adds each binary and supporting constraight to ensure each
    #demand load goes over exactly two different paths
    #first, add that all possible source:destination pairs only use one souce node
    for i in range(1,sources + 1):
        for j in range(1,destination + 1):
            entry = ""
            #now we're in source/destination pairs, find binarys
            for k in range(1,transit+1):
                entry += "u{}{}{} + ".format(i,k,j)
            entry = entry[0:-2]
            entry += "= 2 \n"
            binary_string += entry
    #That first section covers u111 + u121 + u131 type stuff
    #second part is 2 (path) - load (path binary) = 0
    #checks that either both are 0 or both are equal to load dependent
    # on if the binary is true
    #if it's being used, flow = half max
    #otherwise, should be zero
    for i in range(1,sources+1):
        for k in range(1,transit+1):
            for j in range(1,destination+1):
                binary_string += "2 x{}{}{} - {} u{}{}{} = 0 \n".format(i,k,j,i+j,i,k,j)
    return binary_string

def load_balance_r(sources,transit,destination):
    #make the r equations to minimise
    r_string = ""
    for k in range(1,transit+1):
        big_entry = ""
        for i in range(1,sources+1):
            sub_entry = ""
```

```
            for j in range(1,destination+1):
                sub_entry += "x{}{}{} + ".format(i,k,j)
            big_entry += sub_entry
        big_entry = big_entry[0:-2]
        big_entry += "- r <= 0\n"
        r_string += big_entry
    return r_string

def bounds(sources,transit,destination):
    #bounds section (r,path flow,transit to destination and source to transit)
    bound_string = ""

    bound_string += "r >= 0 \n"

    for i in range(1,sources + 1):
        for k in range(1, transit + 1):
            for j in range(1, destination + 1):
                bound_string += "x{}{}{} >= 0 \n".format(i,k,j)

    for k in range(1,transit + 1):
        for i in range(1, sources + 1):
            bound_string += "c{}{} >= 0 \n".format(i,k)

    for k in range(1,transit + 1):
        for j in range(1,destination + 1):
            bound_string += "d{}{} >= 0 \n".format(k,j)

    return bound_string

def binarys(sources,transit,destination):
    binary_string = ""
    for i in range(1,sources + 1):
        for k in range(1,transit + 1):
            for j in range(1, destination + 1):
                binary_string += "u{}{}{} \n".format(i,k,j)

    return binary_string




def main():
    bar = "------------------------------------------------"
    #sources = 3
    #transit = 2
    #destination = 3
    sources = int(sys.argv[1])
    transit = int(sys.argv[2])
    destination = int(sys.argv[3])

    # order mentioned in problem description
    # source to transit capacity,transit to destination capacity,
    # #source to destination demand load, split over 2 paths

    if transit < 2:
        print("Invalid. Transit nodes must number at least 2.")
        sys.exit()

    #start generating the lp file
    lp_file = ""
    lp_file += "Minimize\n"
    lp_file += "r\n"
    lp_file += "Subject to\n"

    #source to transit capacity, cik
    source_cap = source_to_transit_capacity(sources,transit,destination)
    lp_file += source_cap

    #transit to destination capacity, dkj
    transit_cap = transit_to_desination_capacity(sources,transit,destination)
    lp_file += transit_cap

    #source to destination demand load
    demand_load = source_to_dest_demand_volume(sources,transit,destination)
```

```
    lp_file += demand_load

    #split over 2 paths/binary func
    binary_values = split_along_two_paths(sources,transit,destination)
    lp_file += binary_values

    # everything for a given transit -r <= 0
    min_r_line = load_balance_r(sources,transit,destination)
    lp_file += min_r_line

    #bounds section (r,path flow,transit to destination and source to transit)
    lp_file += "Bounds \n"
    bound = bounds(sources,transit,destination)
    lp_file += bound

    #binarys
    lp_file += "Binary \n"
    binary = binarys(sources,transit,destination)
    lp_file += binary

    lp_file += "End"

    with open('323.lp', 'w') as f:
        f.write(lp_file)


main()
```