```
# coding: utf-8

# In[1]:

import pandas as pd
import copy
import numba
from itertools import permutations
from collections import OrderedDict
get_ipython().magic('load_ext autotime')


# In[2]:

game_data = pd.read_excel('/Users/jason.katz/Downloads/Analytics_Attachment.
    xlsx', "2016_17_NBA_Scores")
team_data = pd.read_excel('/Users/jason.katz/Downloads/Analytics_Attachment.
    xlsx', "Division_Info")


# In[3]:

def initialize_team_standings(team_data):
    """
    Provides access to a team's:
        divison
        conference
        head to head record versus every other opponent
        number of games left to be played
        maximum number of wins attainable
        games won
        and maximum number of wins attainable vs conference opponents
    """
    teams = {}
    for _, row in team_data.iterrows():
        teams[row['Team_Name']] = {'Division': row['Division_id'], 'Conference
            ': row['Conference_id'],
                                   'Games_Left': 82, 'Max_Wins': 82, 'Games_Won
                                       ': 0, 'Head2Head': {},
                                   'Conference_Max_Wins': 52}
        for _, row_inside in team_data.iterrows():
            teams[row['Team_Name']]['Head2Head'][row_inside['Team_Name']] = 0
    return teams


# In[4]:

def get_league_data(team_data):
    """
    Provides access to each conference's:
        teams
        divisions
        teams within each division
    """
    east_divisions = set(team_data[team_data['Conference_id'] == 'East']
```

```
        ['Division_id'])
    west_divisions = set(team_data[team_data['Conference_id'] == 'West']
        ['Division_id'])
    east_teams = set(team_data[team_data['Conference_id'] == 'East']['Team_Name
        '])
    west_teams = set(team_data[team_data['Conference_id'] == 'West']['Team_Name
        '])
    return {'East': {'Divisions': east_divisions, 'Teams': east_teams},
            'West': {'Divisions': west_divisions, 'Teams': west_teams}}


# In[5]:

def initialize_league_standings(team_data):
    """
    Provides access to the maximum number of wins among all teams within:
        a conference
        each division in a conference
    """
    league_standings = {'East': {'Conference': {}, 'Atlantic': {}, 'Central':
        {}, 'Southeast': {}},
                        'West': {'Conference': {}, 'Northwest': {}, 'Southwest': {}
                            , 'Pacific': {}}}
    for index, row in team_data.iterrows():
        league_standings[row['Conference_id']]['Conference'][row['Team_Name']]
            = 82
        league_standings[row['Conference_id']][row['Division_id']][row
            ['Team_Name']] = 82
    return league_standings


# In[6]:

class DayOfGames(object):
    def __init__(self, data, team_standings, league_standings):
        self.data = data
        self.date = self.data.iloc[0,0]
        self.games = self.create_game_dicts()
        self.games_simulation = copy.deepcopy(self.games)
        self.add_game_results()
        self.team_standings = copy.deepcopy(DayOfGames.team_standings_class)
        self.league_standings = copy.deepcopy(DayOfGames.league_standings_class
            )
        self.playoff_teams = self.get_playoff_teams()

    def create_game_dicts(self):
        """
        Creates a list of dictionaries for each game with the winning team and
            losing team
        """
        games = []
        for index, row in self.data.iterrows():
            game = {}
            if row['Winner'] == 'Home':
                game[row['Home Team']] = 'Winner'
                game[row['Away Team']] = 'Loser'
```

```
            else:
                game[row['Home Team']] = 'Loser'
                game[row['Away Team']] = 'Winner'
            games.append(game)
        return games

    def add_game_results(self, simulation=False):
        """
        Uses the information from each game to add the results to the current
            standings
        """
        team_standings = DayOfGames.team_standings_class
        league_standings = DayOfGames.league_standings_class
        if simulation:
            games = self.games_simulation
        else:
            games = self.games
        for game in games:
            teams = list(game.keys())
            for idx, team in enumerate(teams):
                team_standings[team]['Games_Left'] -= 1
                if game[team] == 'Loser':
                    team_standings[team]['Max_Wins'] -= 1
                    team_standings[team]['Head2Head'][teams[idx-1]] -= 1
                    if team_standings[team]['Conference'] == team_standings
                        [teams[idx-1]]['Conference']:
                        team_standings[team]['Conference_Max_Wins'] -= 1
                else:
                    team_standings[team]['Games_Won'] += 1
                    team_standings[team]['Head2Head'][teams[idx-1]] += 1
        for team, info in team_standings.items():
            league_standings[info['Conference']]['Conference'][team] = info
                ['Max_Wins']
            league_standings[info['Conference']][info['Division']][team] = info
                ['Max_Wins']

    def simulate_day(self, team_name, win, other_teams=[]):
        """
        For a specific team and desired outcome, goes through all the games and
            changes the outcome if necessary
        """
        for game in self.games_simulation:
            if team_name in game and not [i for i in other_teams if i in game]:
                for team in game:
                    if team == team_name:
                        if win:
                            game[team] = 'Winner'
                        else:
                            game[team] = 'Loser'
                    else:
                        if win:
                            game[team] = 'Loser'
                        else:
                            game[team] = 'Winner'

    def get_division_leaders(self, conference, simulation=True):
```

```
        """
        Creates a list of all teams leading their division
        """
        division_leaders = []
        if simulation:
            league_standings = DayOfGames.league_standings_class
            team_standings = DayOfGames.team_standings_class
        else:
            league_standings = self.league_standings
            team_standings = self.team_standings
        for division in self.league_data[conference]['Divisions']:
            max_wins = 0
            for team, wins in league_standings[conference][division].items():
                if wins > max_wins:
                    leader = team
                    max_wins = wins
                elif wins == max_wins:
                    if team_standings[team]['Head2Head'][leader] > 0:
                        leader = team
            division_leaders.append(leader)
        return division_leaders

    def get_playoff_teams(self, simulation=False, final_day=False):
        """
        Gets the 8 playoff teams for each conference using the division leaders
            and current standings
        """
        if simulation:
            league_standings = DayOfGames.league_standings_class
            team_standings = DayOfGames.team_standings_class
        else:
            league_standings = self.league_standings
            team_standings = self.team_standings
        playoff_teams_all = {}
        for conference in ['East', 'West']:
            division_leaders = self.get_division_leaders(conference, simulation
                )
            for team, max_wins in league_standings[conference]['Conference'].
                items():
                standings = sorted(league_standings[conference]['Conference'],
                                   key=league_standings[conference]['Conference
                                       '].get, reverse=True)
            playoff_teams = [x for x in standings if x not in division_leaders]
                [0:5] + division_leaders
            seed_8 = standings[7]
            seed_9 = standings[8]
            seed_8_info = team_standings[seed_8]
            seed_9_info = team_standings[seed_9]
            if seed_8_info['Max_Wins'] == seed_9_info['Max_Wins']:
                tie_break = seed_8_info['Head2Head'][seed_9]
                if tie_break < 0:
                    playoff_teams.remove(seed_8)
                    playoff_teams.append(seed_9)
                else:
                    if seed_8_info['Conference_Max_Wins'] < seed_9_info
                        ['Conference_Max_Wins']:
```

```
                        playoff_teams.remove(seed_8)
                        playoff_teams.append(seed_9)
            playoff_teams_all[conference] = playoff_teams
        if final_day:
            DayOfGames.final_playoff_teams = playoff_teams_all
        else:
            return playoff_teams_all

    @staticmethod
    def initialize_class_variables():
        DayOfGames.team_standings_class = initialize_team_standings(team_data)
        DayOfGames.league_standings_class = initialize_league_standings
            (team_data)
        DayOfGames.league_data = get_league_data(team_data)
        DayOfGames.days = []

    @staticmethod
    def calculate_first_day_to_check():
        """
        For non playoff teams, calculates the first possible day they are not
            eliminated from the playoffs
        """
        elimination_days = {'East': {}, 'West': {}}
        total_days = len(DayOfGames.days)
        for conference in ['East', 'West']:
            for team in DayOfGames.league_data[conference]['Teams']:
                if team not in DayOfGames.final_playoff_teams[conference]:
                    day_num = total_days
                    eliminated = True
                    while eliminated:
                        day_num -= 1
                        max_wins = DayOfGames.days[day_num].team_standings[team
                            ]['Max_Wins']
                        can_overtake = False
                        for other_team in DayOfGames.days[day_num].
                            playoff_teams[conference]:
                            if max_wins > DayOfGames.days[day_num].
                                team_standings[other_team]['Games_Won']:
                                eliminated = False
                    elimination_days[conference][team] = day_num
        DayOfGames.first_potential_not_eliminated_day = elimination_days

    def reset_game_simulation(self):
        self.games_simulation = copy.deepcopy(self.games)


# In[7]:

def munge_data(team_data, game_data):
    league_data = get_league_data(team_data)
    team_standings = initialize_team_standings(team_data)
    league_standings = initialize_league_standings(team_data)
    first_game_date = game_data.iloc[0,0]
    first_game_index = 0
    DayOfGames.initialize_class_variables()
    for index, row in game_data.iloc[0:].iterrows():
```

```
        if row['Date'] != first_game_date:
            DayOfGames.days.append(DayOfGames(game_data.iloc[first_game_index:
                index], team_standings, league_standings))
            first_game_index = index
            first_game_date = row['Date']
    DayOfGames.days.append(DayOfGames(game_data.iloc[first_game_index:index+1],
        team_standings, league_standings,))
    for idx, day in enumerate(DayOfGames.days):
        setattr(day, 'day', idx)
    DayOfGames.days[-1].get_playoff_teams(final_day=True)
    DayOfGames.calculate_first_day_to_check()


# In[8]:

def run_simulation(day_num, team, win, simulated_teams=[], total_days=162,
    team_to_check=None):
    """
    Simulate a team winning or losing all games from a specified day till the
        end of the season

    Parameters
    ----------
    day_num : Integer
        First day to simulate a team's games
    team : String
        Name of the team to simulate
    win : Boolean
        True for the team winnig all games, False for losing
    simulate_teams : list[String]
        A list of teams which have already had their games simulated (in order
            to prevent overwriting)
    total_days : Integer
        Number of different days with at least one game
    """
    conference = DayOfGames.team_standings_class[team]['Conference']
    DayOfGames.team_standings_class = copy.deepcopy(DayOfGames.days[day_num-1].
        team_standings)
    DayOfGames.league_standings_class = copy.deepcopy(DayOfGames.days[day_num-1
        ].league_standings)
    for day in range(day_num, total_days):
        DayOfGames.days[day].simulate_day(team, win, simulated_teams)
        DayOfGames.days[day].add_game_results(simulation=True)
    simulated_teams.append(team)
    if team_to_check is not None:
        return team_to_check in DayOfGames.days[-1].get_playoff_teams
            (simulation=True)[conference]
    else:
        return team in DayOfGames.days[-1].get_playoff_teams(simulation=True)
            [conference]


# In[9]:

def max_wins_after_simulations(day_num, other_team, simulated_teams):
    """
```

```
    Recalculates a team's maximum wins after some teams' games have been
        simulated
    """
    max_wins = DayOfGames.days[day_num].team_standings[other_team]['Max_Wins']
    for simulated_team in simulated_teams:
        new_head2head = DayOfGames.team_standings_class[other_team]['Head2Head
            '][simulated_team] - DayOfGames.days[130].team_standings[other_team
            ]['Head2Head'][simulated_team]
        if new_head2head < 0:
            max_wins += new_head2head
    return max_wins


# In[10]:

def win_1_lose_1(teams_left, day_num, simulated_teams_copy, team_to_check):
    """
    If two teams are simulated to lose the rest of their games and they play
        each other twice, allows for the
    situation where they go one and one versus each other
    """
    for team_losing in teams_left:
        for other_team in teams_left:
            if team_losing != other_team:
                times_playing_each_other = 0
                for day in DayOfGames.days[day_num:]:
                    for game in day.games:
                        if team_losing in game and other_team in game:
                            times_playing_each_other += 1
                if times_playing_each_other == 2:
                    simulated_teams = copy.deepcopy(simulated_teams_copy)
                    run_simulation(day_num, team_losing, False, simulated_teams
                        , team_to_check=team_to_check)
                    conference = DayOfGames.team_standings_class[team_losing]
                        ['Conference']
                    division = conference = DayOfGames.team_standings_class
                        [team_losing]['Division']
                    DayOfGames.team_standings_class[team_losing]['Max_Wins'] +=
                        1
                    DayOfGames.team_standings_class[team_losing]['Games_Won'] +
                        = 1
                    DayOfGames.team_standings_class[team_losing]['Head2Head']
                        [other_team] += 2
                    DayOfGames.team_standings_class[team_losing]
                        ['Conference_Max_Wins'] += 1
                    DayOfGames.league_standings_class[conference]['Conference']
                        [team_losing] += 1
                    DayOfGames.league_standings_class[conference][Division]
                        [team_losing] += 1
                    division = conference = DayOfGames.team_standings_class
                        [other_team]['Division']
                    DayOfGames.team_standings_class[other_team]['Max_Wins'] -=
                        1
                    DayOfGames.team_standings_class[other_team]['Games_Won'] -=
                        1
                    DayOfGames.team_standings_class[other_team]['Head2Head']
```

```
                        [team_losing] -= 2
                    DayOfGames.team_standings_class[other_team]
                        ['Conference_Max_Wins'] -= 1
                    DayOfGames.league_standings_class[conference]['Conference']
                        [other_team] -= 1
                    DayOfGames.league_standings_class[conference][Division]
                        [other_team] -= 1
                    if team_to_check in DayOfGames.days[-1].get_playoff_teams
                        (simulation=True)[conference]:
                        return True
    return False


# In[11]:

def simulate_teams_to_win(team, day_num, conference, simulated_teams):
    """
    Selects specific teams in a conference and simulates them winnings all
        their remaining games
    Teams who already have more wins than the maximum number of wins for the
        team in question
    Teams with less maximum wins than the team in question

    Parameters
    ----------
    day_num : Integer
        First day to simulate a team's games
    conference : String
        The conference name of the team in question
    max_wins : Integer
        The maximum number of wins the team in question can still attain
    simulate_teams : list[String]
        A list of teams which have already had their games simulated (in order
            to prevent overwriting)
    """
    max_wins = DayOfGames.days[day_num].team_standings[team]['Max_Wins']
    for other_team in DayOfGames.league_data[conference]['Teams']:
        more_wins_than_max = max_wins < DayOfGames.days[day_num].team_standings
            [other_team]['Games_Won']
        less_max_wins = max_wins > max_wins_after_simulations(day_num,
            other_team, simulated_teams)
        if other_team not in simulated_teams and (more_wins_than_max or
            less_max_wins):
            if run_simulation(day_num, other_team, True, simulated_teams,
                team_to_check=team):
                return True
    return False


# In[12]:

def simulate_teams_to_lose(team, day_num, conference, simulated_teams):
    """
    Selects specific teams in a conference and simulates them losing all their
        remaining games
    Teams who are competing for the final playoff spot(s)
```

```
    Iterates over all possible orders for simulating a team losing all their
        games
    Checks if any of the different simulations lead to the team in question
        making the playoffs

    Parameters
    ----------
    team : String
        The team in question
    day_num : Integer
        First day to simulate a team's games
    conference : String
        The conference name of the team in question
    max_wins : Integer
        The maximum number of wins the team in question can still attain
    simulate_teams : list[String]
        A list of teams which have already had their games simulated (in order
            to prevent overwriting)

    Returns
    -------
    ___ : Boolean
        True is the team has a path to the playoffs, False if no path is found,
            meaning they are eliminated
    """
    max_wins = DayOfGames.days[day_num].team_standings[team]['Max_Wins']
    simulated_teams_copy = copy.deepcopy(simulated_teams)
    teams_left = copy.deepcopy(DayOfGames.league_data[conference]['Teams'])
    for simulated_team in simulated_teams_copy:
        teams_left.remove(simulated_team)
    possible_sims = list(permutations(teams_left))
    for sim in possible_sims:
        for other_team in sim:
            if run_simulation(day_num, other_team, False, simulated_teams,
                team_to_check=team):
                return True
            simulated_teams = copy.deepcopy(simulated_teams_copy)
    return win_1_lose_1(teams_left, day_num, simulated_teams_copy, team)


# In[13]:

def get_elimination_days():
    """
    Gets the day each team is eliminated from the playoffs, simulating their
        best case scenario
    """
    total_days = len(DayOfGames.days)
    elimination_day = {}
    for conference in ['East', 'West']:
        for team in DayOfGames.first_potential_not_eliminated_day[conference]:
            if team not in DayOfGames.final_playoff_teams[conference]:
                day_num = DayOfGames.first_potential_not_eliminated_day
                    [conference][team]
                while day_num>=0:
                    for games in DayOfGames.days:
```

```
                        games.reset_game_simulation()
                    simulated_teams = []
                    if run_simulation(day_num, team, True, simulated_teams):
                        break
                    else:
                        if simulate_teams_to_win(team, day_num, conference,
                            simulated_teams):
                            break
                        else:
                            if simulate_teams_to_lose(team, day_num, conference
                                , simulated_teams):
                                break
                    day_num -= 1
                if day_num < 0:
                    print('Something went wrong, team was always found to be
                        eliminated')
                elimination_day[team] = DayOfGames.days[day_num+1].date
    return elimination_day


# In[14]:

munge_data(team_data, game_data)


# In[15]:

elimination_results = pd.DataFrame(OrderedDict({'Team': [''], 'Date Eliminated
    ': ['']}))
results = get_elimination_days()
idx = 0
for team, date in results.items():
    date_str = results[team].strftime("%Y/%m/%d")
    elimination_results.set_value(idx, 'Team', team)
    elimination_results.set_value(idx, 'Date Eliminated', date_str)
    idx+=1
for conference in ['East', 'West']:
    for team in DayOfGames.final_playoff_teams[conference]:
        elimination_results.set_value(idx, 'Team', team)
        elimination_results.set_value(idx, 'Date Eliminated', 'Playoffs')
        idx+=1
elimination_results = elimination_results.sort_values('Team', ascending=True)
elimination_results.to_csv('elimination_results.csv', index = False)
elimination_results


# In[ ]:
```