Untitled

March 20, 2024

1 BitCamp Expo Algorithm Development

By: Sumit Nawathe

```
[78]: import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      import scipy
      import functools
      import itertools
      from typing import List, Dict, Tuple, Optional
```

2 BitCamp Dataset EDA

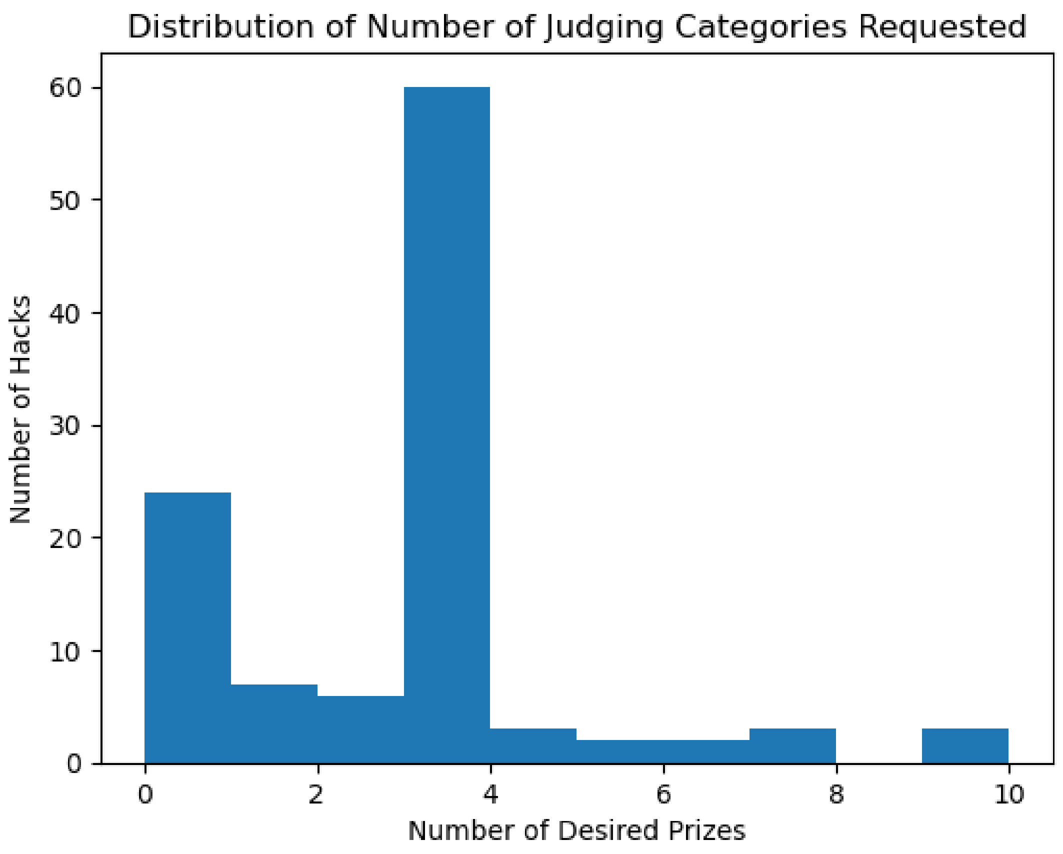
We begin by visualizing the distribution of parameters for previous BitCamp Expos.

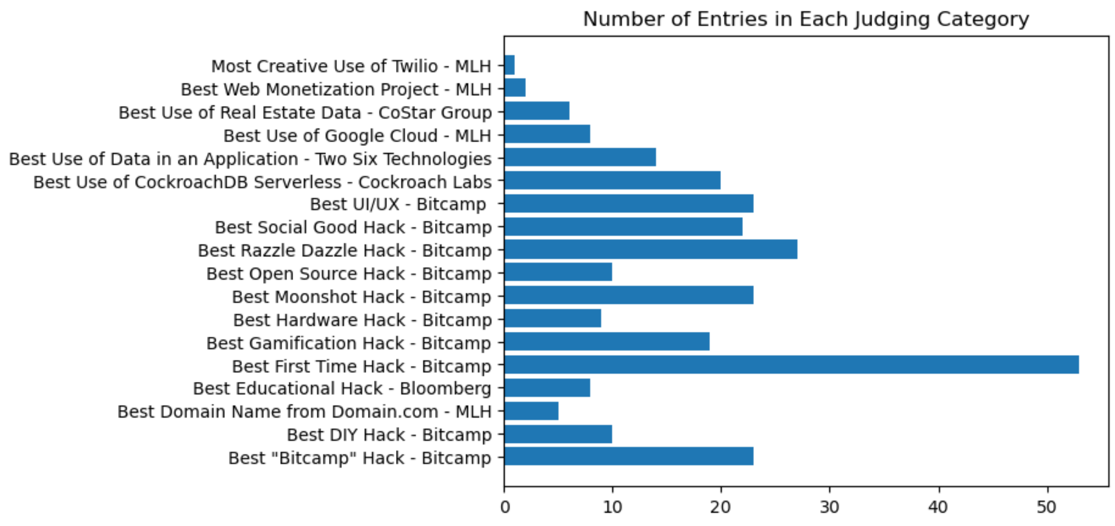
```
[15]: def plot_hacks(hacks: pd.Series):
          num_prizes_by_hack = hacks.to_numpy().sum(axis=1)
          plt.hist(num_prizes_by_hack, bins=max(num_prizes_by_hack))
          plt.xlabel('Number of Desired Prizes')
          plt.ylabel('Number of Hacks')
          plt.title('Distribution of Number of Judging Categories Requested')
          plt.show()

          plt.barh(hacks.columns, hacks.sum())
          plt.title('Number of Entries in Each Judging Category')
          plt.show()
```

```
[16]: ROOT_DIR = "C:/Users/Sumit/GitRepos/Bitcamp/hackathon-expo-app/data/"
```

2.1 2022 Dataset

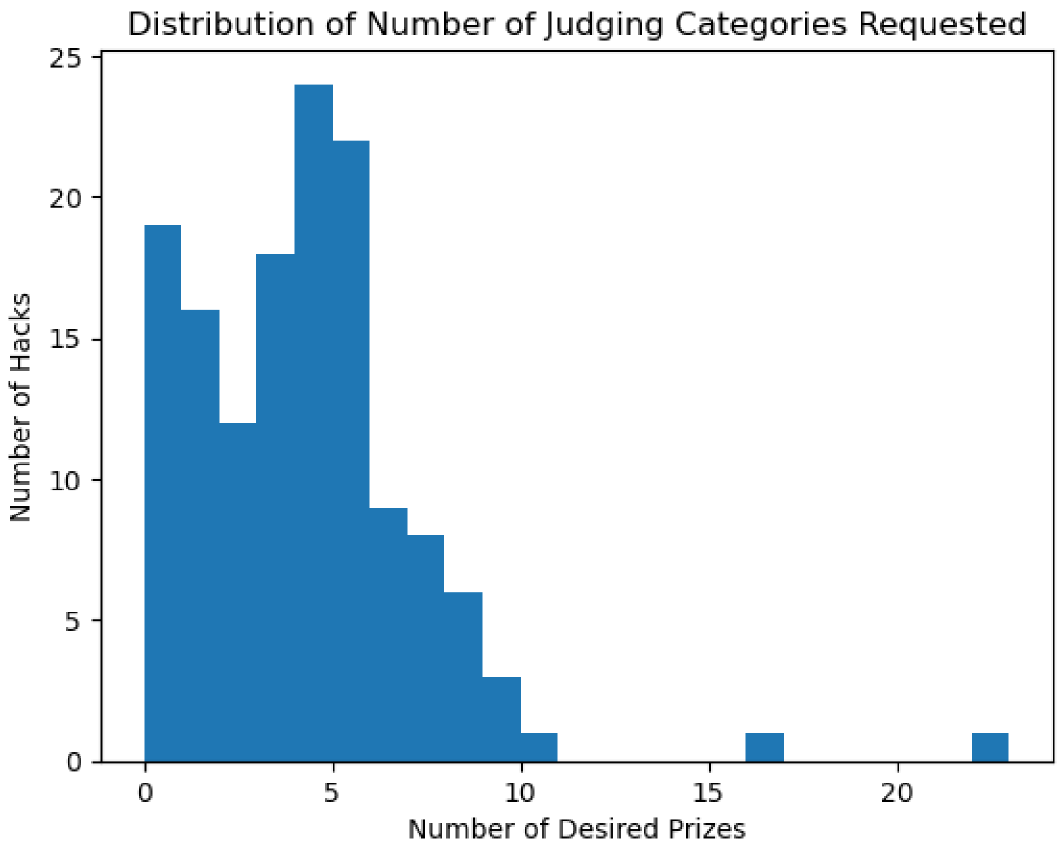
```
[17]: df = pd.read_csv(ROOT_DIR+"bitcamp2022/projects-bitcamp-2022.csv",␣
       ↪dtype='string')
      del df['Judging Status'] # all 'Pending'
      df = df.astype({
          'Project Status': 'category',
          'Highest Step Completed': 'category',
})
hacks = pd.get_dummies(df['Desired Prizes'].str.split(', ').explode()).
 ↪groupby(level=0).sum()
plot_hacks(hacks)
```

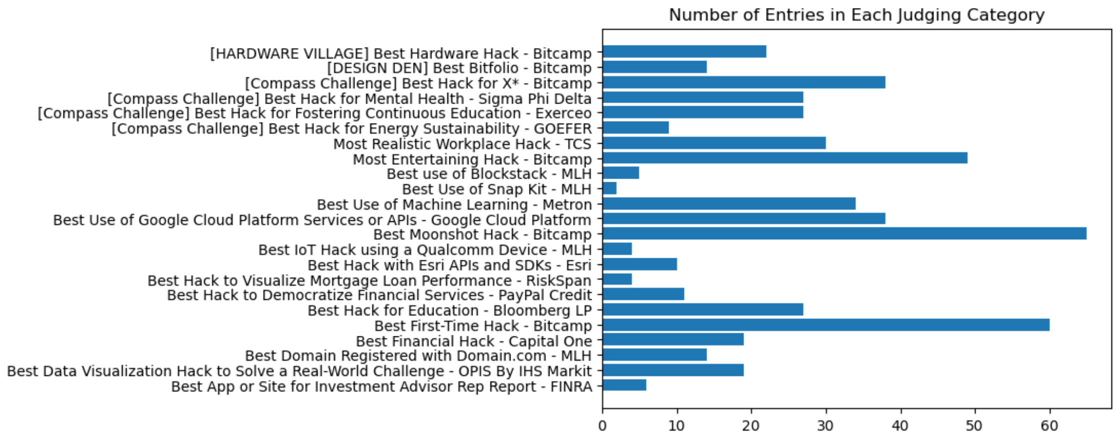




2.2 2019 Dataset

[18]:
```
df = pd.read_csv(ROOT_DIR+"bitcamp2019/bitcamp-2019.csv", dtype='string')
hacks = pd.get_dummies(df['Desired Prizes'].str.split(', ').explode()).
 ↪groupby(level=0).sum()
plot_hacks(hacks)
```





2.3 Comments

Unfortunately, a large number of people try to register for more than three judging categories.

Furthermore, there is a large disparity between the most and least popular judging categories.

From these graphs, we can get decent estimates of the scale of our problem.

3 Algorithm

This algorithm is very abstract, mainly working with the theoretical setup. Much of the preprocessing work must be done before and after.

3.1 Mathematical Description

Suppose there are M hacks, N judging categories, and T time periods. Let $x_{hjt} \in \{0, 1\}$ be a boolean that represents whether hack h is being judged in category j during time period t. We have the following linear constraints (note that the notation $[n] = \{1, 2, \ldots n\}$):

- For every hack, it can be judged at most once in every judging category (ideally exactly once for the categories it signed up for and zero times otherwise).

$$\forall (h, j) \in [M] \times [N], \sum_t x_{hjt} \leq 1$$

- For every judging category, it can judge at most C_j hacks in every time period.

$$\forall (j, t) \in [N] \times [T], \sum_h x_{hjt} \leq C_j$$

- Each hack can only be judged by one category at any time.

$$\forall (h, t) \in [M] \times [T], \sum_j x_{hjt} \leq 1$$

As these are all linear constraints, we can model this problem as an integer linear program, and solve it using the `scipy.optimize.milp` library function.

The number of variables is rather large; in the formulation above, there are MNT variables. We can reduce this by only considering (h, j) pairs that have been registered. Since each hack can only submit to 3 judging categories, this reduces the number of variables to $3MT$. (Note that this restriction is not checked by the algorithm; it is assumed to be true in the provided input.)

3.2 Input/Output Description

Our algorithm takes three arguments:

- `hc: List[List[int]]`. For hack `i`, we set `hc[i]` to be a list of the judging categories that hack `i` enrolls in.
- `cap: List[int]`. For judging category `j`, we set `cap[j]` to be the maximum number of hacks that can be judged from that category every time period.
- `t_max: int`. The largest possible value of `T`. We assume that a schedule exists with this many time periods.

The beginning of the algorithm consists of bookkeeping to properly index all the variables, according to the comment at the end of the "Mathematical Description" section. We flatten the provided `hc` list, and then build a dictionary to map from `(h, j, t)` tuples to variable indices, which we use when constructing the matricies for the linear programming algorithm.

The remainder of the algorithm is running the linear programs within binary search, which is used to find the minimum number of time periods such that there is a valid schedule where everyone is judged (in order to maximize the time per judging).

The outcome of the algorithm is a 3-tuple: * `t: int`. This is the optimal value of `t` via the binary search. * `H: List[List[Tuple[int, int]]]`. We set `H[h]` to be a list of `(j, t)` tuples describing when and by whom hack `h` is being judged. * `J: List[List[List[int]]]`. We let `J[j][t]` be a list of the hacks that category `j` will be judging at time `t`.

3.3 Algorithm Code

[152]:
```
def abstract_expo_alg(hc: List[List[int]], cap: List[int], t_max: int):
    # extracting sizes
    M = len(hc)
    N = len(cap)

    # bookkeeping for valid (h, j) pairs
    valid_hj = set()
    for h, req_cat in enumerate(hc):
        for j in req_cat:
            valid_hj.add((h, j))
    hj_to_i_base = dict(map(tuple, map(lambda t: t[::-1],␣
 ↪list(enumerate(valid_hj)))))

    def solve_expo(T: int):
        # index bookkeeping
        num_var = len(valid_hj) * T
        def hjt_to_i(h: int, j: int, t: int) -> int:
            return len(valid_hj) * (t-1) + hj_to_i_base[(h, j)]

        # first condition
        A1 = np.zeros((len(valid_hj), num_var))
        for x, (h, j) in enumerate(valid_hj):
            for t in range(T):
                A1[x, hjt_to_i(h, j, t)] = 1
        b1= np.ones(len(valid_hj))

        # second condition
        A2 = np.zeros((N*T, num_var))
        for x, (j, t) in enumerate(itertools.product(range(N), range(T))):
            for h in range(M):
                if (h, j) not in valid_hj:
                    continue
                A2[x, hjt_to_i(h, j, t)] = 1
        b2 = np.repeat(cap, T)

        # third condition
        A3 = np.zeros((M*T, num_var))
        for x, (h, t) in enumerate(itertools.product(range(M), range(T))):
            for j in range(N):
                if (h, j) not in valid_hj:
                    continue
                A3[x, hjt_to_i(h, j, t)] = 1
        b3 = np.ones(M*T)

        # solve linear program
        x = scipy.optimize.milp(
            c=-np.ones(num_var),
            constraints=[
                scipy.optimize.LinearConstraint(A1, 0, b1),
                scipy.optimize.LinearConstraint(A2, 0, b2),
                scipy.optimize.LinearConstraint(A3, 0, b3)
            ],
            bounds=scipy.optimize.Bounds(lb=0, ub=1),
            integrality=1
        ).x
        if int(sum(x)) < len(valid_hj):
            return None

        # interpret solution
        H = [list() for _ in range(M)]
        J = [list() for _ in range(N)]
        for j in range(N):
            J[j] = [list() for _ in range(T)]
            for h in range(M):
                if (h, j) not in valid_hj:
                    continue
                for t in range(T):
                    if x[hjt_to_i(h, j, t)] == 1.0:
                        H[h].append((j, t))
                        J[j][t].append(h)
        return (H, J)

    # return solve_expo(t_max)
    # binary search:
    a, b = 1, t_max
    while a < b-1:
        m = int(np.ceil((a+b)/2))
        soln = solve_expo(m)
        if soln is None: # failure
            a = m+1
        else: # success
            b = m

    # check when 2 left
    if a == b:
        t = a
    else:
        if solve_expo(a) is None:
            t = b
        else:
            t = a

    # return optimal solution
    H, J = solve_expo(t)
    return (t, H, J)
```

4 Tests

4.1 Simple Example 1

Suppose we have a hackathon with 3 contestants and 3 judges.

- Hack 1 wants to be judged in categories 1 and 2
- Hack 2 wants to be judged in categories 2 and 3
- Hack 3 wants to be judged in category 3

Supose that categories 1 and 2 can judge 1 person at a time, and category 3 can judge 2 people at a time.

Clearly, this hackathon only requires 2 time periods; running the algorithm yields this result.

(NOTE: remember that everything is 0-indexed.)

[153]:
```
hc = [
    [0, 1],
    [1, 2],
    [2]
]
cap = [1, 1, 2]
```

[154]:
```
abstract_expo_alg(hc, cap, 5)
```

[154]:
```
(2,
 [[(0, 1), (1, 0)], [(1, 1), (2, 0)], [(2, 1)]],
 [[[], [0]], [[0], [1]], [[1], [2]]])
```

4.2 Simple Example 2

Suppose we have a hackathon with 5 contestants and 3 judges.

- Hack 1 wants to be judged in categories 1, 2, and 3
- Hack 2 wants to be judged in categories 1, 2, and 4
- Hack 3 wants to be judged in categories 2, 5
- Hack 4 wants to be judged in categories, 1, 2, and 3
- Hack 5 wants to be judged in categories 1 and 3
- Hack 6 wants to be judged in categories, 1, 2, and 3

Supose that the limits for judging categories 1, 2, 3, 4, 5 are 2, 2, 1, 3, 1, respectively.

The most popular categories can judge fairly quickly (categories 1 and 2 each have 4 participants, but can judge 2 at a time). We cannot finish judging in 2 time steps, because most of the hacks need to be judged in 3 categories. However, unfortunately categories 1, 2, and 3 are so popular that they cannot jointly cover everyone in 3 time steps with the restrictions; 4 time steps are necessary.

[160]:
```
hc = [
    [0, 1, 2],
    [0, 1, 3],
    [1, 4],
    [0, 1, 2],
    [0, 2],
    [0, 1, 2]
]
cap = [2, 2, 1, 3]
```

[161]:
```
abstract_expo_alg(hc, cap, 5)
```

[161]:
```
(4,
 [[(0, 2), (1, 3), (2, 1)],
  [(0, 3), (1, 1), (3, 0)],
  [(1, 2)],
  [(0, 3), (1, 1), (2, 0)],
  [(0, 0), (2, 3)],
  [(0, 1), (1, 3), (2, 2)]],
 [[[4], [5], [0], [1, 3]],
  [[], [1, 3], [2], [0, 5]],
  [[3], [0], [5], [4]],
  [[1], [], [], []]]])
```

[ ]: