In [1]:
```
from instrument import instrument
```

6.009 Tutorial 7 -- Who's Got Talent?

Introduction

We are auditioning candidates for a show called "Who's got Talent?" Each candidate is capable of performing a set of talents. Some candidates might be capable of performing multiple talents (e.g., Singing, Dancing, Humor, etc.), while others might be able to perform only one or none at all. In this lab you will write an algorithm for selecting which candidates to perform on the show. Every selected candidate will perform all of his or her talents. The producer has given you two constraints.

First, the producer wants "Who's Got Talent?" to draw a wide audience -- this means incorporating a wide range of talents into the show. You will be given a set of talents that the producer cares about. Every one of these talents must be performed in the show.

Second, the producer wants to limit costs, which are proportional to the number of performers. You need to select as few performers as possible.

Thus your ultimate goal is to select the fewest possible number of performers, while still ensuring that every talent that the producer cares about will be performed.

Problem Setup -- with an Example

There are multiple ways to represent our universe of candidates and talents. For your convenience we provide a list-of-list representations, `candidate_to_talents`, the list of talents that Candidate `c` is able to perform. **For convenience, we assume that the producer wants to include every talent in the show.** (This is not much of an assumption, because otherwise we could define our universe to only include the talents that the producer cares about.)

For example, consider the following candidate-talent universe, where "x" denotes the ability of a candidate:

Candidate	Talent 0	Talent 1	Talent 2	Talent 3
Candidate 0	x	x	x	x
Candidate 1		x		x
Candidate 2		x	x	

In our representation, `candidate_to_talents` is the following list of lists:

```
candidate_to_talents[0] = [0, 1, 2, 3]
candidate_to_talents[1] = [1, 3]
candidate_to_talents[2] = [1, 2]
```

Implementation -- select_candidates

Inputs

- num_candidates: number of candidates; candidates are numbered 0 to (num_candidates - 1)
- num_talents: number of talents; talents are numbered 0 to (num_talents - 1)
- candidate_to_talents: list of lists; candidates_to_talents[c] is a list of talents that candidate c is able to perform

Output

- list of candidate indices, of minimal length to cover the talents. If no solution exists, return an empty list [].

Brute Force Approach

Idea: try *all* combinations of candidates, and pick the one that covers all of the talents, with the smallest number of candidates.

In [2]:
```
def select_candidates(num_candidates, num_talents,
                      candidate_to_talents):
    # use sets for fast intersections and differences
    c_to_t = [set(candidate_to_talents[c])
                  for c in range(num_candidates)]

    # does set of candidates cover all the talents?
    def covers(cset):
        talents = set()
        for c in cset:
            talents.update(c_to_t[c])
        return len(talents) == num_talents

    # fancy pythonic equivalent using argument unpacking
    def covers(cset):
        return len(set().union(*[c_to_t[c] for c in cset])) == num_talents

    call_count = 0    #just instrumentation

    # look at all possible candidate sets; pick best (smallest) one
    all_candidate_sets = all_subsets(list(range(num_candidates)))
    best = None
    best_count = num_candidates + 1
    for cset in all_candidate_sets:
        call_count += 1
        if covers(cset):
            if len(cset) < best_count:
                best_count = len(cset)
                best = list(cset)
    print("num_candidates:", num_candidates,
          "; cases considered:", call_count, "; best:", best)
    return best or []
```

In [3]:
```
# yield all subsets of list L; all elements of L assumed to be unique
def all_subsets(L):
    if len(L) == 0:
        yield set()
    else:
        first = set([L[0]])
        for s in all_subsets(L[1:]):
            yield s
            yield first | s
```

In [4]:
```
list(all_subsets([1,2,3]))
```

Out[4]:
```
[set(), {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Visualization Utility

A utility to help us visualize talents (horizontal) for each candidate (vertical)

In [5]:
```
def get_matrix(num_candidates, num_talents, candidate_to_talents):
    return [[1 if t in candidate_to_talents[c] else 0 \
             for t in range(num_talents)] for c in range(num_candidates)]
```

In [6]:
```
def print_matrix(m):
    print("              Talents ->")
    for r in range(len(m)):
        print("Candidate {0:2d}:".format(r), m[r])
```

Our Example

In [7]:
```
num_candidates = 3
num_talents = 4
candidate_to_talents = [[0, 1, 2, 3], [1, 3], [1, 2]]
print_matrix(get_matrix(num_candidates, num_talents, candidate_to_talents))
```

```
              Talents ->
Candidate  0: [1, 1, 1, 1]
Candidate  1: [0, 1, 0, 1]
Candidate  2: [0, 1, 1, 0]
```

In [8]:
```
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
```

```
num_candidates: 3 ; cases considered: 8 ; best: [0]
```

Out[8]: [0]

Test 1: Small, Simple Case

In [9]:
```
num_candidates = 3
num_talents = 3
candidate_to_talents = [[0], [1], [2]]
print_matrix(get_matrix(num_candidates, num_talents,
                        candidate_to_talents))
```

```
              Talents ->
Candidate  0: [1, 0, 0]
Candidate  1: [0, 1, 0]
Candidate  2: [0, 0, 1]
```

In [10]:
```
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
```

```
num_candidates: 3 ; cases considered: 8 ; best: [0, 1, 2]
```

Out[10]: [0, 1, 2]

Test 2: Large, Simple Case

In [11]:
```
num_candidates = 10
num_talents = 10
candidate_to_talents = [[0], [1], [2], [3], [4], [5], [6], [7], [8],
                        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
print_matrix(get_matrix(num_candidates, num_talents,
                        candidate_to_talents))
```

```
              Talents ->
Candidate  0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate  1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate  2: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
Candidate  3: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Candidate  4: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
Candidate  5: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
Candidate  6: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
Candidate  7: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
Candidate  8: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
Candidate  9: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [12]:
```
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
```

```
num_candidates: 10 ; cases considered: 1024 ; best: [9]
```

Out[12]: [9]

In [13]:
```
print(2**num_candidates)
```

```
1024
```

Test 3: No Solution case

In [14]:
```
num_candidates = 4
num_talents = 5
candidate_to_talents = [[0], [1], [2], [3]]
print_matrix(get_matrix(num_candidates, num_talents,
                        candidate_to_talents))
```

```
              Talents ->
Candidate  0: [1, 0, 0, 0, 0]
Candidate  1: [0, 1, 0, 0, 0]
Candidate  2: [0, 0, 1, 0, 0]
Candidate  3: [0, 0, 0, 1, 0]
```

In [15]:
```
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
print(2**num_candidates)
```

```
num_candidates: 4 ; cases considered: 16 ; best: None
16
```

Test 6: Non-Greedy case

In [16]:
```
num_candidates = 4
num_talents = 6
candidate_to_talents = [[0, 1], [2, 3], [4, 5], [0, 2, 4]]
print_matrix(get_matrix(num_candidates, num_talents,
                        candidate_to_talents))
```

```
              Talents ->
Candidate  0: [1, 1, 0, 0, 0, 0]
Candidate  1: [0, 0, 1, 1, 0, 0]
Candidate  2: [0, 0, 0, 0, 1, 1]
Candidate  3: [1, 0, 1, 0, 1, 0]
```

If we take Candidate 3, we'll *still* need the other three to cover all of the talents. So a 'greedy' approach successively grabbing the candidate with the most talent until all talents are filled would fail to find the smallest group of candidates...

In [17]:
```
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
print(2**num_candidates)
```

```
num_candidates: 4 ; cases considered: 16 ; best: [0, 1, 2]
16
```

Test 7: Big Non-Greedy case

In [18]:
```
num_candidates = 17
num_talents = 18
candidate_to_talents = [[0, 3, 6], [3, 4, 5], [6, 7, 8], [9, 10, 11],
                        [12, 13, 14], [15, 16, 17], [0, 1, 2], [17],
                        [1, 4], [7, 10], [13, 16], [2], [5], [8], [11],
                        [14], [9, 12, 15]]
print_matrix(get_matrix(num_candidates, num_talents,
                        candidate_to_talents))
```

```
               Talents ->
Candidate  0: [1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate  1: [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate  2: [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate  3: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0]
Candidate  4: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
Candidate  5: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1]
Candidate  6: [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate  7: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
Candidate  8: [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate  9: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
Candidate 10: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0]
Candidate 11: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate 12: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate 13: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Candidate 14: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Candidate 15: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
Candidate 16: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
```

Candidates 1 through 6 cover our talents; but if we take Candidate 0, we still need Candidates 1 through 6. So our algorithm should decline to take Candidate 0...

In [19]:
```
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
print(2**num_candidates)
```

```
num_candidates: 17 ; cases considered: 131072 ; best: [1, 2, 3, 4, 5, 6]
131072
```

Better version

In the brute force approach, we consider *every* grouping of candidates. But what if the very first candidate satisfies all of the talents? That would mean a lot of wasted work to look at all the other cases!

More generally, we'd like a better version of `select_candidates` that doesn't waste time on cases that can't be better than the best we've found so far -- i.e., if we're just adding candidates to the chosen list, but the chosen list being considered is already larger in size than the best previously found best list of candidates.

In [20]:
```
def select_candidates(num_candidates, num_talents,
                      candidate_to_talents):
    call_count = 0

    # use sets for fast intersections and differences
    c_to_t = [set(candidate_to_talents[c]) for c in range(num_candidates)]

    # return best (smallest) set of candidates that provide the needed talents
    #  needed_talents = set of talents needed
    #  available = list of available candidates
    #  chosen = list of candidates chosen so far
    #  best = best solution found so far (None == no solution)
    #@instrument
    def helper(needed_talents, available, chosen, best):
        nonlocal call_count
        call_count += 1

        # have we found a solution?  [base case]
        if len(needed_talents) == 0:
            return chosen if best is None or len(chosen) < len(best) else best

        # optimization #1: we already know a better solution
        if best is not None and len(chosen) >= len(best): return best

        # optimization #2: only consider candidates with at least one needed talent
        available = [c for c in available if c_to_t[c] & needed_talents]

        # are we out of candidates?  [base case]
        if len(available) == 0: return best

        # see which works best: either using the first candidate or not
        first = available[0]   # trial candidate
        rest = available[1:]   # remaining candidates

        # try solution without using trial candidate [recursive case]
        if rest:
            best = helper(needed_talents, rest, chosen, best)

        # try solution using trial candidate [recursive case]
        best = helper(needed_talents - c_to_t[first], rest, chosen + [first], best)

        return best

    # initially we need all the talents, all candidates are available,
    # no one has been chosen, and we have no best solution
    needed_talents = set(range(num_talents))
    available = list(range(num_candidates))
    result = helper(needed_talents, available, [], None)

    print("num_candidates:", num_candidates, "; num recursive calls:", call_count, "; best:", result)

    # put result in desired form
    return result or []
```

Now we can go back and run our test cases... and see if they indeed require looking at fewer combinations of candidates!

Test 1: Small, Simple Case (revisited, showing recursion call structure)

In [21]:
```
num_candidates = 3
num_talents = 3
candidate_to_talents = [[0], [1], [2]]
print_matrix(get_matrix(num_candidates, num_talents,
                        candidate_to_talents))
```

```
             Talents ->
Candidate  0: [1, 0, 0]
Candidate  1: [0, 1, 0]
Candidate  2: [0, 0, 1]
```

In [22]:
```
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
print(2**num_candidates)
```

```
num_candidates: 3 ; num recursive calls: 11 ; best: [0, 1, 2]
8
```

Some additional things to try

- See how the recursion progresses: try with instrumented version (uncomment `@instrument`)
- Change the order of try solution with and without trial candidate; how does that change the recursion?
- What is the complexity of the search?
- How might we implement a breadth first search (i.e., to try all single candidate choices, followed by all two-candidate choices, etc.)?

Brute force breadth-first

In [23]:
```
## Try sets of candidates in order of size of sets (smaller sets first).
## By definition, the first one we find will be minimal and be a best result.
def bfbf_select_candidates(num_candidates, num_talents,
                           candidate_to_talents):
    # use sets for fast intersections and differences
    c_to_t = [set(candidate_to_talents[c]) for c in range(num_candidates)]

    def covers(candidates):
        talents = set()
        for c in candidates:
            talents.update(c_to_t[c])
        return len(talents) == num_talents

    candidates = list(range(num_candidates))
    case_count = 0
    for size in range(1,num_candidates+1):
        for candidate_set in all_subsets_of_size(candidates, size):
            case_count += 1
            if covers(candidate_set):
                print("num_candidates:", num_candidates, " num cases:", case_count, "best:", candidate_set)
                return list(candidate_set)
    print("num_candidates:", num_candidates, " num cases:", case_count, "best:", [])
    return []
```

In [24]:
```
# yield all subsets of L equal in size to size
def all_subsets_of_size(L, size):
    if len(L) < size:
        return
    if size == 0:
        yield set()
        return
    first = L[0]
    rest = L[1:]
    yield from all_subsets_of_size(rest, size)
    for subset in all_subsets_of_size(rest, size-1):
        subset.add(first)
        yield subset

print(list(all_subsets_of_size([1,2,3], 2)))
#[{2, 3}, {1, 3}, {1, 2}]
```

```
[{2, 3}, {1, 3}, {1, 2}]
```

In [25]:
```
# yield all subsets of L equal in size to size
def all_subsets_of_size(L, size):
    if len(L) < size:
        return
    if size == 0:
        yield set()
        return
    for first in range(len(L) - size + 1):
        for subset in all_subsets_of_size(L[first+1:], size-1):
            subset.add(L[first])
            yield subset

print(list(all_subsets_of_size([1,2,3], 2)))
#[{2, 3}, {1, 3}, {1, 2}]
```

```
[{1, 2}, {1, 3}, {2, 3}]
```

This above is conceptually simple... but is still somewhat slow. The timing test below can be used to quantify (and compare to our other implementations).

Why is this somewhat slow? We look at reasonable numbers of combinations, but we've ignored the cost of creating the combinations (using `all_subsets` or `all_subsets_of_size`). In this case, rebuilding sets of different sizes from scratch on each set size iteration can be costly. It also recalculates talent coverage for each new candidate set from scratch.

So we might want to explore ways to speed those up, or better yet, to structure a select_candidates recursion that builds candidate cases from smaller to larger.

In [26]:
```
import time
num_candidates = 17
num_talents = 18
candidate_to_talents = [[0, 3, 6], [3, 4, 5], [6, 7, 8], [9, 10, 11],
                        [12, 13, 14], [15, 16, 17], [0, 1, 2], [17],
                        [1, 4], [7, 10], [13, 16], [2], [5], [8], [11],
                        [14], [9, 12, 15]]
start = time.time()
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
end = time.time()
print("time:", end-start, "sec")
print(2**num_candidates)
```

```
num_candidates: 17 ; num recursive calls: 21360 ; best: [1, 2, 3, 4, 5, 6]
time: 0.04472970962524414 sec
131072
```

In [27]:
```
# BIG CASE for timing -- need ALL candidates
import time
num_candidates = 20
num_talents = num_candidates
candidate_to_talents = [[c] for c in range(num_candidates)]
#print_matrix(get_matrix(num_candidates, num_talents,
#                        candidate_to_talents))

start = time.time()
select_candidates(num_candidates, num_talents,
                  candidate_to_talents)
end = time.time()
print("time:", end-start, "sec")
print(2**num_candidates)
```

```
num_candidates: 20 ; num recursive calls: 1572863 ; best: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1
6, 17, 18, 19]
time: 4.282547235488892 sec
1048576
```

Efficiency

The `select_candidates` task is an example of the *set cover* problem, which is NP-hard. That is to say, there is no known polynomial time algorithm -- and thus the problem quickly becomes intractable for larger size. Above, we see that the number of calls grows as 2^n (exponentially). Try increasing from 20 candidates above, to 21, to 22, etc., and see how the time increases quickly!