John Luke Denny

Problem:
A company wishes to purchase n items. Various suppliers, totaling k, offer packages of items that are a subset of $\{a_1, a_2, \ldots, a_n\}$. These packages must be bought in whole at a cost C. The goal of the company is to minimize the cost spent getting at least one of each item. Once one of each item is bought, the company need not buy any more packages.

Optimal solution:
An optimal solution to this problem is to iterate all possible combinations to find the lowest cost combination that has all of the items. This will always be optimal because all combinations are checked. However, this is not computationally good because there will be 2^k non-repetative combinations. This is way too many to feasibly check, so it might help to create a greedy algorithm.

Heuristic:

1. Sort the packages in ascending order of cost and purchase packages starting at the lowest cost and buying more as long as they contain a new item.
   Pseudocode:

```
sort_by_cost(arr_packages)

Total_items = n

Cost
Suppliers[]
items[]
For package in arr_packages
    If package is not subset of items[]
        Cost += package.cost
        suppliers.add(package.supplier)
        Items = items union package.items
        If length(items) == total_items
            Break;
```

   This pseudocode is a computation of the heuristic. It starts off with a sort to organize the packages. Then, the total number of items, n, is stored in total_items. A cost variable, suppliers array, and items array are initialized. Then, a for-loop is used to iterate through all of the packages. If the package has an item that hasn't been bought yet, the package is bought, the suppliers is track, the items are added via union, and the cost is properly increased. Once all of the items are bought, there is no need to look at anymore packages, so the for-loop is aborted. Given modern sorting algorithms are in O(nlogn) and for-loops are of time O(n), the time complexity of this algorithm is O(klogk + k), simplified to O(klogk), because there are k packages to sort and iterate through. A small optimization can be made if packages with the same cost are sorted based on which

ones have more items. With this heuristic, it focuses on the lowest cost packages, so any time the heuristic breaks out, the cost is a sum of minimal costs. This isn't guaranteed to be optimal, but it does direct it toward smaller costs.

Examples:
Optimal:
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost $C_1 = 14$.
Supplier #2 offers: $\{a_1, a_2, a_8, a_9, a_{10}\}$ at cost $C_2 = 30$.
Supplier #3 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost $C_3 = 12$.

Sorted::
Supplier #3 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost $C_3 = 12$.
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost $C_1 = 14$.
Supplier #2 offers: $\{a_1, a_2, a_8, a_9, a_{10}\}$ at cost $C_2 = 30$.

Total_items = 12
Iter1::
Cost = 12
Suppliers = [supplier 3]
Items = $[a_8, a_9, a_{10}, a_{11}, a_{12}]$
5 != total_items

Iter2::
Cost = 26
Suppliers = [supplier 3, supplier 1]
Items = $[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}]$
12 == total_items

This is optimal because there are no redundancies in the packages pulled.

Nonoptimal:
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost $C_1 = 14$.
Supplier #2 offers: $\{a_1, a_2, a_8, a_9, a_{10}\}$ at cost $C_2 = 12$.
Supplier #3 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost $C_3 = 30$.

Sorted::
Supplier #2 offers: $\{a_1, a_2, a_8, a_9, a_{10}\}$ at cost $C_2 = 12$.
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost $C_1 = 14$.
Supplier #3 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost $C_3 = 30$.

Total_items = 12
Iter1::
Cost = 12
Suppliers = [supplier 2]

Items = [a_1, a_2, a_8, a_9, $a_{10,}$]
5 != total_items

Iter2::
Cost = 26
Suppliers = [supplier 2, supplier 1]
Items = [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}]
10 == total_items

Iter3::
Cost = 56
Suppliers = [supplier 2, supplier 1, supplier 3]
Items = [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}]
12 == total_items
Break

This is not optimal because there are redundancies in the packages.

2. Keep track of a variable alongside each package called num_items, this will keep track of how many items the package contains. Sort the packages in descending order based on this metric. Buy packages with the most items as long as there are new items within the package. The goal of this heuristic is to minimize the number of packages bought which usually can help to minimize cost.
   Pseudocode:

```
sort_by_num_items(arr_packages)

Total_items = n

Cost
Suppliers[]
items[]
For package in arr_packages
    If package is not subset of items[]
        Cost += package.cost
        suppliers.add(package.supplier)
        Items = items union package.items
        If length(items) == total_items
            Break;
```

   This heuristic effectively works like the other one in terms of the checks performed and its time complexity. However, this one varies because the basis for the sort is different. A small optimization by organizing ties by ascending cost. With this heuristic, it focuses on the packages with the most items, so any time the heuristic breaks out, the number of

packages summed is minimized. This isn't guaranteed to be optimal, but it does direct it toward smaller costs.

Examples:
Optimal:
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost C_1 = 14. Num_items=7
Supplier #2 offers: $\{a_1, a_2, a_8, a_9\}$ at cost C_2 = 12. Num_items=4
Supplier #3 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost C_3 = 30. Num_items=5

sorted::
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost C_1 = 14. Num_items=7
Supplier #3 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost C_3 = 30. Num_items=5
Supplier #2 offers: $\{a_1, a_2, a_8, a_9\}$ at cost C_2 = 12. Num_items=4

Total_items = 12
Iter1::
Cost = 14
Suppliers = [supplier 1]
Items = [a_1, a_2, a_3, a_4, a_5, a_6, $a_{7,}$]
7 != total_items

Iter2::
Cost = 44
Suppliers = [supplier 1, supplier 3]
Items = [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}]
12 == total_items
Break

This is optimal because supplier 2 is redundant to both supplier 3 and supplier 1.

Nonoptimal:
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost C_1 = 14. Num_items=7
Supplier #2 offers: $\{a_1, a_2, a_8, a_9\}$ at cost C_2 = 12. Num_items=4
Supplier #3 offers: $\{a_3, a_4, a_{10}, a_{11}, a_{12}\}$ at cost C_3 = 25. Num_items=5
Supplier #4 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost C_3 = 30. Num_items=5

Sorted::
Supplier #1 offers: $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ at cost C_1 = 14. Num_items=7
Supplier #3 offers: $\{a_3, a_4, a_{10}, a_{11}, a_{12}\}$ at cost C_3 = 25. Num_items=5
Supplier #4 offers: $\{a_8, a_9, a_{10}, a_{11}, a_{12}\}$ at cost C_3 = 30. Num_items=5
Supplier #2 offers: $\{a_1, a_2, a_8, a_9\}$ at cost C_2 = 12. Num_items=4

Total_items = 12
Iter1::

Cost = 14
Suppliers = [supplier 1]
Items = [a_1, a_2, a_3, a_4, a_5, a_6, $a_{7,}$]
7 != total_items

Iter2::
Cost = 39
Suppliers = [supplier 1, supplier 3]
Items = [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_{10}, a_{11}, a_{12}]
10 != total_items

Iter3::
Cost = 69
Suppliers = [supplier 1, supplier 3, supplier 4]
Items = [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}]
12 == total_items
Break

The optimal cost is still 44 with just using supplier 1 and 4. However, due to redundancy in the packages, this heuristic won't always return optimally.

Other Applications:
With this type of problem, it can be applied to any that are looking to minimize a union of sets to optimize some value. It might be useful in social pairings and maximize happiness, assuming the pairings are sets and the value to optimize is happiness. This could be doneusing the sort by cost heuristics and then filter out the pairings that are redundant to maximize pairing happiness.