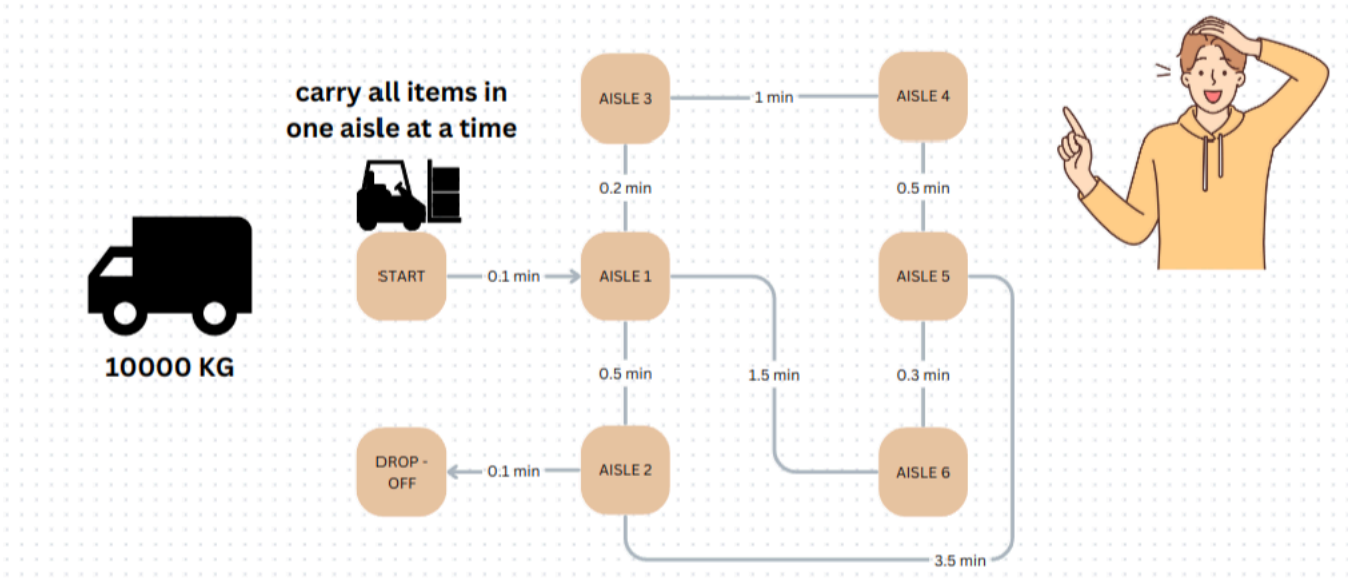


✓ CASE STUDY

- You're an appliances supplier and you want to ship the items from your warehouse to Give to your distributor and sell is As soon as Possible. You have a Forklift that can carry all items in one aisle at a time, You also has a truck that has 10000kg capacity weight which is the storage of your items picked. You have a warehouse that has 6 aisles with different items. Your goal is to get the most preferable items in an aisle and putting it to the truck considering the how many minutes you consume in the process. (Assuming the minutes are constant and picking time is not considered)



- This is the list of the Items in the Warehouse, it has the aisle number and the weight and price as needed to sucessfully run a knapsack approach

NAME	AISLE	WEIGHT	PRICE
set-n-forget cooker	1	1458	2949
washing machine	3	1408	2584
thermal mass refrigerator	5	1313	1686
sous-vide cooker	3	751	2958
electric water boiler	1	1302	4590
energy regulator	4	646	4129
thermal immersion circulator	2	715	3656
air conditioner	6	957	2389
energy regulator	1	598	564
internet refrigerator	6	1193	2361
clothes dryer	2	1079	2671
turkey fryer	1	973	4791
beverage opener	6	1002	3812
home server	4	934	4148
vacuum cleaner	5	1346	864
humidifier	4	792	2690
panini sandwich grill	2	1241	3564
flattop grill	4	1352	1545
microwave oven	1	616	4390
icebox	3	1275	2405
		20951	58746

✓ ANSWER

This problem can be solved by using Knapsack techniques, Graphs, and the use of Dynamic prograaming. In this we started in creating this guidelines and steps of Algorithms.

ALGORITHMS

1. **START**
2. **PERFORM KNAPSACK TO THE ITEMS TO ABLE TO PUT IN THE TRUCK**
3. **SAVE THE KNAPSACK AS IT SERVES AS THE LIST FOR YOUR TRAVEL IN THE WAREHOUSE**
4. **CREATE A GRAPH THAT IS SAME TO THE FIGURE ABOVE**
5. **IDENTIFY THE FASTEST WAY TO GET ALL THE ITEMS**
6. **OUTPUT THE ITEMS NEEDED TO GET AND ORDER OF PATH TO GET IT.**

WITH THIS ALGORITM, WE CAN START THE CODING PROCESS...

- **First, Lets Create a class tthat will handle the items**

```
1 class Items:
2     def __init__(self, name, aisle, price, weight): # Parameters for the Name(str), aisle(int),price(int),and weight(int)
3         self.__name = name
4         self.__aisle = aisle
5         self.__price = price
6         self.__weight = weight
7
8     def get_name(self): # It gets the the encapsulated name
9         return self.__name
10
11     def get_aisle(self): # It gets the the encapsulated aisle
12         return self.__aisle
13
14     def get_price(self): # It gets the the encapsulated price
15         return self.__price
16
17     def get_weight(self): # It gets the the encapsulated weight
18         return self.__weight
```

- **We also need to create a menu list so that all of the Items are organized well**

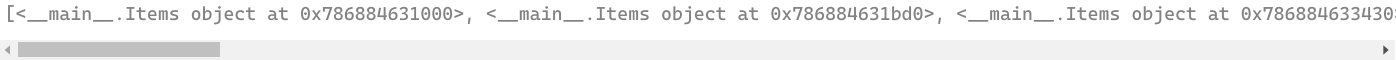
```
1 def menu_of_items(arrOfNames, arrOfAisle,arrOfPrice,arrOfWeight):
2     items = []
3     for i in range(len(arrOfNames)):
4         items.append(Items(arrOfNames[i], arrOfAisle[i], arrOfPrice[i], arrOfWeight[i]))
5     return items
```

- **Now We made inndividual array per category but using the Class and the function, we can store it efficiently.**

```
1 # list of the items in the Warehouse
2 items = ["set-n-forget cooker", "washing machine", "thermal mass refrigerator", "sous-vide cooker", "electric water boiler",
3         "energy regulator", "thermal immersion circulator","air conditioner", "energy regulator", "internet refrigerator",
4         "clothes dryer", "turkey fryer", "beverage opener", "home server", "vacuum cleaner",
5         "humidifier", "panini sandwich grill","flattop grill","microwave oven","icebox"]
6
7 aisle = [1,3,5,3,1,
8         4,2,6,1,6,
9         2,1,6,4,5,
10        4,2,4,1,3]
11
12 price = [2949,2584,1686,2958,4590,
13         4129,3656,2389,564,2361,
14         2671,4791,3812,4148,864,
15         2690,3564,1545,4390,2405]
16
17 weight = [1458,1408,1313,751,1302,
18         646,715,957,598,1193,
19         1079,973,1002,934,1346,
20         792,1241,1352,616,1275]
21
22 capacity = 10000 # for the capacity of the truck
```

- **Lets Feed in the data**

```
1 item_menu = menu_of_items(items, aisle, price, weight) # This variable will be sent to the kanpsack function
2
3 print(item_menu) # Checking the Items in the List
```



- **Let's Now Create a knapsack function for the data and the truck, and this also stores the aisle of the items to be picked**

```
1 def tab_knapsack(cap,items): # it accepts the Item class list we created
2     n = len(items)
3     table = [[0 for i in range(cap+1)] for i in range(n+1)] # the use of tabulation Method
4     for i in range(n+1):
5         for j in range(cap+1):
6             if i == 0 or j == 0:
7                 table[i][j] = 0
8             elif items[i-1].get_weight() ≤ j: # the the capacity can handle the weight of an item, then add to the table
9                 table[i][j] = max(items[i-1].get_price() + table[i-1][j - items[i-1].get_weight()], table[i-1][j])
10
11 k = n
12 l = cap
13 while k > 0 and l > 0: # this loop for for analyzing the table populated with our data and print the items included or not
14     if table[k][l] == table[k-1][l]:
15         items.remove(items[k-1]) # It removes the item that is not Included to the knapsack
16         k-=1
17     else:
18         k-=1
19         l-=items[k].get_weight()
20
21 items.sort(key=lambda x: x.get_aisle()) # sorting the items by Aisle
22 aisles = []
23 for i in items:
24     print(f"{i.get_name()}, is in aisle {i.get_aisle()}, {i.get_weight()}") # Prints the items you include for the truck
25     aisles.append(i.get_aisle())
26 return list(set(aisles)) # Returns the list of the aisles that the items is in
```

- **Now let's Proceed to feeding the items_menu to the knapsack**

```
1 aisle = tab_knapsack(capacity, item_menu) # Display the Included Items, Their Aisle Num and the Weight (For Checking).
```

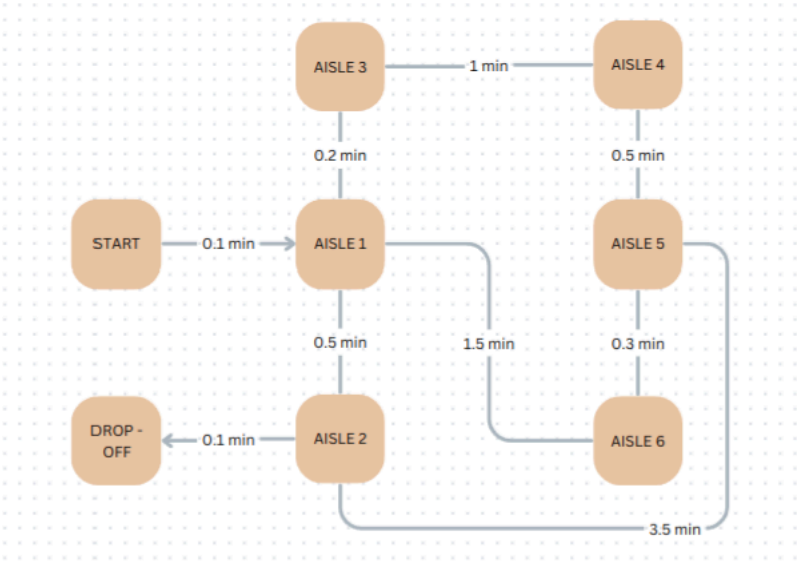
electric water boiler, is in aisle 1, 1302
turkey fryer, is in aisle 1, 973
microwave oven, is in aisle 1, 616
thermal immersion circulator, is in aisle 2, 715
panini sandwich grill, is in aisle 2, 1241
sous-vide cooker, is in aisle 3, 751
energy regulator, is in aisle 4, 646
home server, is in aisle 4, 934
humidifier, is in aisle 4, 792
air conditioner, is in aisle 6, 957
beverage opener, is in aisle 6, 1002

- Checking the List of the aisle needed to be past through.

```
1 print(aisle)

[1, 2, 3, 4, 6]
```

- **Now lets Proceed to the graph part...**



- **We decided to take the simple route in making graph, example is creating a dictionary to represent the graph and corresponding time in connecting edges**

```
1 # STRUCTURE:
2 # { VERTICES: (EDGES, CORRESPONDING TIME FOR EDGE)}
3 g = { "START" : [(1,0.1)],
4       1 : [(3,0.2),(2,0.5),(6,1.5)],
5       2 : [(1, 0.5), (5, 3.5), ("DROP OFF", 0.1)],
6       3 :[(1, 0.2], (4, 1)],
7       4 : [(3,1),(5, 0.5)],
8       5 : [(2, 3.5),(6, 0.3)],
9       6 : [(1, 1.5),(5,0.3)],
10      "DROP OFF":[]
11    }
```

- **Let's Create a Class that Implements this graph**

```
1 class Graph:
2     def __init__(self, graph_dict=None): # It accepts parameters of a graph dictionary, if theres None it creates One.
3         """ initializes a graph object
4             If no dictionary or None is given,
5             an empty dictionary will be used
6         """
7         if graph_dict == None:
8             graph_dict = {}
9         self._graph_dict = graph_dict
10
11     def edges(self, vertice): # Gets all the Edges of a certain vertice
12         """ returns a list of all the edges of a vertice"""
13         return [i for i in self._graph_dict[vertice]]
14
15     def __generate_edges(self):
16         """ A static method generating the edges of the
17             graph "graph". Edges are represented as sets
18             with one (a loop back to the vertex) or two
19             vertices
20         """
21         edges = []
22         for vertex in self._graph_dict:
23             for neighbour in self._graph_dict[vertex]:
24                 if {neighbour, vertex} not in edges:
25                     edges.append({vertex, neighbour})
26         return edges
27
28     def find_all_paths(self, start_vertex, end_vertex, path=[], sums=0):
29         """ find all paths from start_vertex to
30             end_vertex in graph """
31         graph = self._graph_dict
32         path = path + [start_vertex]
33         if start_vertex == end_vertex: # THE BASE CASE, when this is satisfied, it returns all the posible paths, additional th
34             return path, sums
35         if start_vertex not in graph: # If vertex is not their, then give them nothing
36             return []
37         paths = []
38         for vertex in graph[start_vertex]:
39             if vertex[0] not in path:
40                 extended_paths = self.find_all_paths(vertex[0], end_vertex, path, sums + vertex[1]) # RECURSIVE
41                 for p in extended_paths:
42                     paths.append(p)
43                 self.sums = 0
44         return paths
45
46     def total_mins(self, paths):
47         records = {}
48         for i in range(len(paths)):
49             if type(paths[i]) == list: # If the current Item is array, then it is recorded as a value of the path index (path id
50                 records[len(records)] = [paths[i],paths[i+1]]
51         return records
52
53     def fastest_way(self, aisles, total_min):
54         table = [[0 for i in range(len(total_min))] for i in range(len(aisles))]
55         for i in range(len(aisles)):
56             for j in range(len(total_min)):
57                 if aisles[i] in total_min[j][0]:
58                     table[i][j] += total_min[j][1]
59
60         minimum_routes = []
61         print(table)
62         for i in table:
63             smallest_time = min(num for num in i if num > 0)
64             s = i.index(smallest_time)
65             minimum_routes.append(s)
66
67         return minimum_routes
```

• **Let's Add One additional Essiantial Codes**

```
1 """
2     def edges(self, vertice): # Gets all the Edges of a certain vertice
3         """ returns a list of all the edges of a vertice""
4         return [i for i in self._graph_dict[vertice]]
5
6     def __generate_edges(self):
7         "" A static method generating the edges of the
8             graph "graph". Edges are represented as sets
9             with one (a loop back to the vertex) or two
10            vertices
11         ""
12         edges = []
13         for vertex in self._graph_dict:
14             for neighbour in self._graph_dict[vertex]:
15                 if {neighbour, vertex} not in edges:
16                     edges.append({vertex, neighbour})
17         return edges
18         """
19
20     '\n def edges(self, vertice): # Gets all the Edges of a certain vertice\n     "" r
21     eturns a list of all the edges of a vertice""\n         return [i for i in self._graph
22     _dict[vertice]]\n\n def __generate_edges(self): \n         "" A static method generati
23     na the edoes of the \n             araph "araph". Edoes are reopresented as sets \n
```

• **So The First goal is to Identify all the Possible routes from START to the DROP OFF point. so We are going to add a recursive function to find the routes.**

```
1 """ def find_all_paths(self, start_vertex, end_vertex, path=[], sums=0):
```

```
2     """find all paths from start_vertex to
3     end_vertex in graph """
4     graph = self._graph_dict
5     path = path + [start_vertex]
6     if start_vertex == end_vertex: # THE BASE CASE, when this is satisfied, it returns all the posible paths, additional the
7         return path, sums
8     if start_vertex not in graph: # If vertex is not their, then give them nothing
9         return []
10    paths = []
11    for vertex in graph[start_vertex]:
12        if vertex[0] not in path:
13            extended_paths = self.find_all_paths(vertex[0], end_vertex, path, sums + vertex[1]) # RECURSIVE
14            for p in extended_paths:
15                paths.append(p)
16            self.sums = 0
17    return paths"""
```

```
' def find_all_paths(self, start_vertex, end_vertex, path=[], sums=0):\n    """find all paths from start_vertex to \n    e\n    nd_vertex in graph ""\n    graph = self._graph_dict \n    path = path + [start_vertex]\n    if start_vertex == end_vertex:\n    # THE BASE CASE, when this is satisfied, it returns all the posible paths, additional the total time of the path to be finished\n    (based on the time in the edges)\n    return path, sums\n    if start_vertex not in graph: # If vertex is not their, then\n    give them nothina\n    return []\n    paths = []\n    for vertex in graph[start vertex]:\n    if vertex[0] not in p
```

- **With The Output of the finding all paths, we need to fixed it to make us understand more the process, so that this function fixes this paths to be a dictionary, noting that this paths are key for noting the paths.**

```
1     """
2     def total_mins(self, paths):
3         records = {}
4         for i in range(len(paths)):
5             if type(paths[i]) == list: # If the current Item is array, then it is recorded as a value of the path index (path ider
6                 records[len(records)] = [paths[i],paths[i+1]]
7         return records
8     """
```

```
\ndef total_mins(self, paths):\n    records = {}\n    for i in range(len(paths)):\n        if type(paths[i]) == list: # If the\n    current Item is array, then it is recorded as a value of the path index (path identified by the find_all_paths())\n    records[len(records)] = [paths[i], paths[i+1]]\n    return records\n    '
```

- ****And Now we will find the fastest way to get all the items in the using the minimal value in the array. (Greedy Method- getting the Lowest possible time possible)**

```
1     """
2     def fastest_way(self, aisles, total_min):
3         table = [[0 for i in range(len(total_min))] for i in range(len(aisles))]
4         for i in range(len(aisles)):
5             for j in range(len(total_min)):
6                 if aisles[i] in total_min[j][0]:
7                     table[i][j] += total_min[j][1]
8
9         minimum_routes = []
10        print(table)
11        for i in table:
12            smallest_time = min(num for num in i if num > 0)
13            s = i.index(smallest_time)
14            minimum_routes.append(s)
15
16        return minimum_routes
17    """
```

```
\n def fastest_way(self, aisles, total_min):\n    table = [[0 for i in range(len(total_min))] for i in range(len(aisles))]\n    \n    for i in range(len(aisles)):\n        for j in range(len(total_min)):\n            if aisles[i] in total_min[j][0]:\n    table[i][j] += total_min[j][1]\n    \n    minimum_routes = []\n    print(table)\n    for i in table:\n        smallest_\n    time = min(num for num in i if num > 0)\n        s = i.index(smallest time)\n        minimum routes.append(s)\n    \n    return m
```

✓ For the Results, we will be provided a fastest order of path getting items in a particular Aisle.

```
1 graph = Graph(g)
2 path = graph.find_all_paths('START', 'DROP OFF') # All Possible Paths
3 dicxts = graph.total_mins(path)
4 print(dicxts) # The Dictionary Created
5 print(graph.fastest_way(aisle, dicxts)) # The Order of the fastest way to get all the items in the warehouse

{0: [['START', 1, 3, 4, 5, 2, 'DROP OFF'], 5.3999999999999995], 1: [['START', 1, 2, 'DROP OFF'], 0.7], 2: [['START', 1, 6, 5, 2\n[[5.3999999999999995, 0.7, 5.5], [5.3999999999999995, 0.7, 5.5], [5.3999999999999995, 0, 0], [5.3999999999999995, 0, 0], [0, 0,\n[1, 1, 0, 0, 2]
```

