۱. مدل کردن مسئله به این صورت است که نقطه شروع (initial state) مکان قرار گرفتن سید است و نقطه هدف/پایان (goal state) زمانی است که تمامی دستور های پخت جمع آوری شده باشد و دیزی ها به مریدان رسیده باشد. برای حل این مسئله به اینصورت عمل میکنیم که ورودی های داده شده را با کلاسی به اسم Node تعریف کرده و برای هر راس مرید بودن/نبودن، دستورپخت بودن/نبودن، صعب العبور بودن/نبودن، لوکیشن و در صورت مرید بودن، رسپی های آن مرید را ذخیره میکنیم. سپس برای هر راس یک استیت را بصورتی در نظر میگیریم که اطلاعاتی نظیر والد، لوکیشن، هزینه، زمان باقی مانده، مرید های دیده شده، دستورپخت های دیده شده و رئوس صعب العبور دیده شده را ذخیره میکند. در هر استیتی که قرار داریم، استیت های اطراف آن که فرزندان آن استیت را تشکیل میدهند را میسازیم و اطلاعات آنرا با توجه به اطلاعات پدر آن و کلاس نود آن آپدیت میکنیم. به همین صورت تا زمانیکه به استیت هدف برسیم ادامه میدهیم.

Initial state: محل قرارگیری سید

Action: حرکت به نود های همسایه (نودهایی که نود فعلی با یال به آنها متصل است)

Transition model: حرکات ما فقط باعث رفتن به نودهای همسایه میشود، پس نتیجه هر حرکت جابجایی به یکی از نودهای همسایه است

Goal state: رساندن دیزی ها به تمامی مریدان که برابر این است که که به ازای تمامی مریدان، تمامی رسپی های آن دیده شود.

Path cost: هزینه جابجایی از هر راس برابر یک است، اما اگر راس صعب العبور باشد، هزینه به تعداد دفعاتی که از آن راس عبور میکنیم است، درنتیجه هزینه نهایی برابر مجموع هزینه هر راس است.

۲. توضیح هر الگوریتم:

الگوریتم BFS

در این الگوریتم سطح به سطح پیش میرویم و از صف برای پیاده سازی آن استفاده میکنیم. منظور از جستجوی سطحی این است که هر نود میتواند نود های همسایه خودرا ببیند و آنهارا در صف قرار میدهد. سپس در مرحله نودهای موجود در صف pop شده و همین جستجویی سطحی برای آنها نیز تکرار میشود. از آنجایی که در این الگوریتم سطح به سطح چک میکنیم، بنابراین در داده های بزرگ بدلیل این قدم به قدم پیش رفتن، زمان زیادی برای حل تست گرفته میشود. بنابراین میتوان نتیجه گرفت این الگوریتم برای داده های کوچک میتواند مناسب باشد، اما در داده های بزرگ ضعیف عمل میکند.

نحوه پیاده سازی ما با توجه به صورت سوال به این شکل است که در ابتدا نود شروع (start_node) را به صف های frontier و explored اضافه میکنیم. سپس چک میکنیم که آیا این نود goal state ما هست یا نه. اگر بود که مسیر را چاپ کرده و در غیر اینصورت وارد حلقه میشویم. شرط حلقه while این است که تا زمانی که

frontier پر باشد و همچنین به goal state نرسیده باشیم، از حلقه خارج نمیشویم. در ابتدا نود را از صف pop کرده و اگر نود صعب العبور با زمان بزرگتر از صفر نبود، همسایه نود را بررسی میکنیم بصورتیکه در ابتدا این همسایه اطلاعات پدر خودرا به ارث میبرد و سپس هزینه‌مان را افزایش میدهیم. در مرحله بعدی چک میکنیم که نود فرزند در صف explored وجود دارد یا نه و سپس تست goal را بررسی میکنیم. اما اگر نود، صعب العبور با زمان بزرگتر از صفر بود، به اندازه آن باید در آن راس بمانیم، بنابراین هزینه را به اندازه یک واحد افزایش داده و زمان باقی مانده برای آن راس را یک واحد کاهش میدهیم و سپس نود را به صف اضافه میکنیم. در نهایت زمانی که به goal رسیدیم، مسیر طی شده را چاپ میکنیم.

```
def algorithm(self, start):
  start_node = State(n = self.size, state = start, morids = morids)
  self.counter = 1
  explored = []
  isReachedGoal = False
  self.updateChild(start_node, start)
  explored.append(start_node)
  goal = start_node
  self.add(start_node)
  if(start_node.goalTest()):
        isReachedGoal = True
  while (self.is_empty() and not isReachedGoal):
      node = self.remove()
      if node.remainingTime == 0:
          for neighbor in self.graph[node.state]:
              child_node = State(state=neighbor, n=self.size, morids = self.morids)
              child_node.copyFromParent(node)
              self.updateChild(child_node, neighbor)
              self.counter += 1
              child_node.increaseCost()
              if not self.contains(child_node, explored):
                  if(child_node.goalTest()):
                      goal = child_node
                      isReachedGoal = True
                      break
                  explored.append(child_node)
                  self.add(child_node)
      else:
          node.decreaseRemainingTime()
          node.increaseCost()
          self.add(node)

  self.printPath(goal)
```

الگوریتم IDS

در این الگوریتم بر خلاف الگوریتم قبلی که یک الگوریتم جستجوی سطحی بود، به صورت عمیق پیش میرویم و از استک برای پیاده سازی آن استفاده میکنیم. منظور از جستجوی عمیق آن است که هر نود تا اخرین فرزند نود همسایه خودرا طی میکند و هربار آنرا وارد استک کرده و به سراغ فرزند میرود. این الگوریتم مشابه الگوریتم DFS است با این تفاوت که یک عمق محدودی در نظر میگیریم و DFS را اجرا میکنیم، اگر پاسخ به دست آمد که آنرا چاپ میکنیم در غیر اینصورت، الگوریتم را با عمق بیشتر تکرار میکنیم و اینکار را تا زمانی که به جواب برسیم ادامه میدهیم.

نحوه پیاده سازی ما با توجه به صورت سوال به این شکل است که در ابتدا با عمق یک الگوریتم را صدا میزنیم. سپس نود شروع (start_node) را تعریف کرده و تابع بازگشتی را صدا میزنیم. در این تابع ابتدا تست goal را انجام میدهیم، اگر به goal state رسیده بودیم، مسیر را چاپ کرده و برنامه پایان می باید، در غیر اینصورت اگر عمق به صفر رسیده بود اما هنوز به نود هدف نرسیده بودیم فالس برگردانده شده و عمق را افزایش میدهیم، در غیر این دو صورت نود های همسایه را چک کرده و بصورت بازگشتی اینکار را تا زمانیکه به جواب برسیم تکرار میکنیم.

```
def recursiveDLS(self, node, maxDepth):
    if(node.goalTest()):
        goal = node
        isReachedGoal = True
        return goal, isReachedGoal

    if maxDepth == 0:
        return None, False
    isReachedGoal = False
    if node.remainingTime == 0:
        for neighbor in self.graph[node.state]:
            child_node = State(state=neighbor, n=self.size, morids = self.morids)
            child_node.copyFromParent(node)
            self.updateChild(child_node, neighbor)
            self.counter += 1
            child_node.increaseCost()
            res, isReachedGoal = self.recursiveDLS(child_node, maxDepth - 1)
            if isReachedGoal:
                return res, True
    else:
        node.decreaseRemainingTime()
    return None, isReachedGoal

  def DLS(self,start,maxDepth):
    start_node = State(n = self.size, state = start, morids = morids)
    self.updateChild(start_node, start)
    self.counter = 1
    path, isReachGoal =  self.recursiveDLS(start_node, maxDepth)
    if isReachGoal:
        self.prirtPath(path)
        return True
    return False

  def IDS(self, start):
    i = 1
    while(True):
        if (self.DLS(start, i)):
            break
        i = 1
```

الگوریتم A*

الگوریتم a* یک الگوریتم مسیریابی است که بدلیل بهینه بودن و داشتن سرعت مناسب از آن استفاده میشود. در این الگوریتم از heap استفاده میکنیم. پیاده سازی به این صورت است که ما یک تخمین هزینه برای رسیدن از نود فعلی به نود هدف داریم که آنرا با h(n) مشخص میکنیم(باید توجه داشت که پیدا کردن یک هیوریستیک مناسب بر روی سرعت برنامه تاثیر دارد.) همچنین هزینه ای که تا الان برای رسیدن به این نود طی شده را با g(n) نشان میدهیم و هزینه نهایی که از جمع h(n) و g(n) بدست می آید را با f(n) مشخص میکنیم. هربار باید هزینه رسیدن به نودی که در آن قرار داریم را با هزینه ای که قبلا به این نود رسیدیم چک کنیم تا مطمئن شویم که هزینه ای که در نهایت در لیستمان قرار میگیرد، مینیموم باشد.

نحوه پیاده سازی ما با توجه به صورت سوال به این شکل است که در ابتدا نود شروع (start_node) را به صف های frontier و explored اضافه میکنیم. در این الگوریتم چون مقدار f(n) هر نود برای ما اهمیت دارد، بنابراین اولویت قرارگیری نودها در آرایه نیز باید بر اساس این مولفه باشد. در مرحله بعد چک میکنیم که آیا این نود goal state ما هست یا نه. اگر بود که مسیر را چاپ کرده و در غیر اینصورت وارد حلقه میشویم. شرط حلقه while این است که تا زمانی که frontier پر باشد و همچنین به goal state نرسیده باشیم، از حلقه خارج نمیشویم. در ابتدای حلقه آرایه frontier را heapify کرده و نود را از آن pop میکنیم. اگر نود صعب العبور با زمان بزرگتر از صفر نبود، همسایه نود را بررسی میکنیم بصورتیکه در ابتدا این همسایه اطلاعات پدر خودرا به ارث میبرد و سپس هزینه‌مان را افزایش داده و تست goal را بررسی میکنیم. پس از آن مقدار های g(n)، h(n) و f(n) را محاسبه کرده و شروط گفته شده در توضیحات الگوریتم را بررسی میکنیم. اما اگر نود، صعب العبور با زمان بزرگتر از صفر بود، به اندازه آن باید در آن راس بمانیم، بنابراین هزینه را به اندازه یک واحد افزایش داده و زمان باقی مانده برای آن راس را یک واحد کاهش میدهیم و سپس نود را به صف اضافه میکنیم. در نهایت زمانی که به goal رسیدیم، مسیر طی شده را چاپ میکنیم.

```
def algorithm(self, start):
    explored = []
    frontier = []
    isReachedGoal = False
    start_node = State(n = self.size, state = start, morids = morids)
    self.counter = 1
    frontier.append((start_node.f, start_node))
    explored.append(start_node)
    goal = start_node
    self.updateChild(start_node, start)
    if(start_node.goalTest()):
        isReachedGoal = True

    while (frontier and not isReachedGoal):
        heapq.heapify(frontier)
        node = heapq.heappop(frontier)
        nodeF = node[0]
        node = node[1]
        if node.remainingTime == 0:
            for neighbor in self.graph[node.state]:
                child_node = State(state=neighbor, n=self.size, morids = self.morids)
                child_node.copyFromParent(node)
                self.updateChild(child_node, neighbor)
                self.counter += 1
                child_node.increaseCost()
                if(child_node.goalTest()):
                    goal = child_node
                    isReachedGoal = True
                    break
                nodeInfo = allNodes[neighbor]
                child_node.g = node.g + child_node.cost
                child_node.heuristic(neighbor, nodeInfo.moridRecipes, recipes)
                child_node.h = child_node.costToGoal
                child_node.f = child_node.g + child_node.h * ALPHA

                if not self.contains(child_node, explored):
                    explored.append(child_node)
                    heapq.heappush(frontier, (child_node.f, child_node))

        else:
            node.decreaseRemainingTime()
            node.increaseCost()
            frontier.append((child_node.f, child_node))
    self.printPath(goal)
```

۳. Heuristic از مجموع رسپی های دیده نشده و مرید های دیده نشده را بدست می آید. Consistent بودن heuristic نیز به اینصورت تعریف میشود که اگر فرض کنیم مرید و رسپی دیده نشده در نزدیکترین استیت با کمترین هزینه رسیدن به آن قرار دارند، فقط یک واحد به costGoal اضافه میشود که به سادگی مشخص میشود که این هزینه کمتر مساوی هزینه حقیقی است. در غیر اینصورت اگر فرض کنیم که مرید و رسپی دیده نشده در استیت دیگری باشند، میتوان فرض کرد که هزینه رسیدن از یک نود(۱) به نود هدف برابر α و هزینه یکی از اجدادش(۲) تا نود هدف برابر β باشد، همچنین طبق تعریف heuristic مجموع هزینه مرید و رسپی دیده نشده در هر یک از نود های ۱ و ۲ کمتر مساوی α و β خواهد بود، بنابرین با در نظر گرفتن هزینه ای که نود های واسط پرداخت میکنند، میتوان نتیجه گرفت که:

Cost(1 to 2) >= h(1) - h(2)

```
def heuristic(self, s, moridsRec, recipe):
    temp = []
    for i in range(len(recipe)):
        if recipe[i]:
            if not self.seenRecipes[i]:
                self.costToGoal += 1
                temp.append(i)
    for info in self.seenMorids:
        if not self.seenMorids[info] and int(info) not in temp:
            self.costToGoal += 1
```

.۴

تست اول	پاسخ مسئله	تعداد استیت های دیده شده	میانگین زمان اجرا
BFS	۸	۱۵۰	۰/۰۰۴
IDS	۸	۲۴۷۶	۰/۰۱
A*	۸	۱۳۷	۰/۰۰۲
Weighted A*1	۸	۱۳۷	۰/۰۰۸
Weighted A*2	۸	۱۳۷	۰/۰۰۵

BFS

```
Count: 150
Cost: 8
1->3->4->5->7->10->11->9->8
time : 0.0010211467742919922
```

IDS

```
Count: 2476
cost: 8
1->3->4->5->7->10->11->9->8
time : 0.011811256408691406
```

A*

```
Count: 137
Cost: 8
1->3->4->5->7->10->11->9->8
time : 0.0025420188903808594
```

Weighted A*: ALPHA = 1.3

```
Count: 137
Cost: 8
1->3->4->5->7->10->11->9->8
time : 0.008055686950683594
```

Weighted A*: ALPHA = 1.6

```
Count: 137
Cost: 8
1->3->4->5->7->10->11->9->8
time : 0.0039904117584228516
```

تست دوم	پاسخ مسئله	تعداد استیت های دیده شده	میانگین زمان اجرا
BFS	۱۲	۴۰۱۹۴	۸
IDS			
A*	۱۲	۳۲۹۶۴	۷
Weighted A*1	۱۲	۳۲۹۶۵	۱۰
Weighted A*2	۱۲	۳۲۹۶۵	۱۰

BFS

```
Count: 40194
Cost: 12
28->19->13->3->11->24->9->23->28->23->5->7->29
time : 8.87961196899414
```

A*

```
Count: 32964
Cost: 12
28->30->9->24->11->3->13->23->5->7->29->22->28
time : 7.3021135330200195
```

Weighted A*: ALPHA = 1.3

```
Count: 32965
Cost: 12
28->23->13->3->11->24->9->22->28->23->5->7->29
time : 10.422191381454468
```

Weighted A*: ALPHA = 1.6

```
Count: 32962
Cost: 12
28->23->13->3->11->24->9->2->5->7->29->22->28
time : 10.185092449188232
```

تست سوم	پاسخ مسئله	تعداد استیت های دیده شده	میانگین زمان اجرا
BFS	۲۱	۳۲۴۶۲	۱۰
IDS			
A*	۲۱	۱۵۶۰۵	۳
Weighted A*1	۲۱	۱۵۶۱۳	۳
Weighted A*2	۲۱	۱۵۶۱۳	۳

BFS

```
Count: 32462
Cost: 21
40->42->38->24->31->45->30->48->41->18->1->19->43->49->47->49->9->34->25->50->12->16
time : 10.119738817214966
```

A*

```
Count: 15605
Cost: 21
40->42->38->24->31->45->30->48->41->18->1->19->43->49->47->49->9->34->25->50->12->16
time : 3.0155975818634033
```

Weighted A*: ALPHA = 1.3

```
Count: 15613
Cost: 21
40->42->38->24->31->45->30->48->41->18->1->19->43->49->47->49->9->34->25->50->12->16
time : 4.513288736343384
```

Weighted A*: ALPHA = 1.6

```
Count: 15613
Cost: 21
40->42->38->24->31->45->30->48->41->18->1->19->43->49->47->49->9->34->25->50->12->16
time : 3.3721184730529785
```

تست easy 2	پاسخ مسئله	تعداد استیت های دیده شده	میانگین زمان اجرا
IDS	۸	۵۹۸۵	۰/۰۳

```
Count: 5985
cost: 8
9->10->9->4->12->3->7->5->8
time : 0.03207993507385254
```

تست easy 3	پاسخ مسئله	تعداد استیت های دیده شده	میانگین زمان اجرا
IDS	۱۳	۹۱۱۷۹	۰/۶

```
Count: 91179
cost: 13
13->11->10->3->2->6->12->5->9->4->1->13->11->10
time : 0.5194752216339111
```