سوال ۱) الف) برای جواب به این سوال در ژنتیک باید به این نکته توجه کرد که ما در هر مرحله دو والد را crossover می کنیم و یک نسل بهتر می سازیم.حال چون N=1 است یعنی جمعیت فقط یکی است و ما فقط یک والد داریم. پس عمل crossover فقط با همین تک والد است.به زبانی دیگر دو والدی که crossover می کنیم یکی هستند و در نتیجه هر ترکیبی از دو تا چیز یکسان نهایت همان چیزیکسان را می دهد. پس عمل crossover ما را به وضعیت بهتری نمی برد.اما جهش در ژنتیک نیز ممکن است رخ دهد و تنها حالتی که ما را به وضعیت دیگری می برد جهش است.اینجا دو حالت داریم:

- اگر جهش ما همواره به سمت بهتر شدن برود: در این حالت ما با رندم جهش دادن به جواب بهینه محلی نزدیک می شویم که همان صخره نوردی (به طور دقیق تر صخره نوردی تصادفی چون لزوما بهترین حالت بعدی رو انتخاب نمی کنیم و فقط یک حالت بهتر رو انتخاب میکنیم)است.
- اگر قانونی برای جهش نباشد در اینصورت مثل الگوریتم random walk باید بین وضعیت ها جا بجا شویم تا برسیم به جواب.

ب) الگوریتم Local Beam Search همان الگوریتم Hill-Climbing است با این تفاوت که در هر مرحله به جای یک استیت، k استیت داریم. حال اگر k=1 باشد به این معنی است همان hill-Climbing را داریم اجرا میکنیم..

ج) این اتفاق امکان پذیر نیست معادل است با الگوریتم breath first search.ایده به این شکل است که اگر هر جانشین حفظ شود (زیرا k نامحدود است)، آنگاه جستجو مانند BFS است که یک سطر کامل از گره ها را قبل از افزودن سطر بعدی اضافه می کند. با شروع از یک حالت، الگوریتم اساساً با BFS یکسان خواهد بود با این تفاوت که هر سطر به یکباره تولید می شود.

د) در الگوریتم SA اگر آن قسمت که وضعیت دلتا منفی است را حذف کنیم تبدیل به یک hill climbing میشود زیرا برای وضعیت های بد در بهینه محلی گیر میکنند. حال با T=0 احتمالی که در وضعیت بد یک استیت را انتخاب میکند ۰ میشود و امکان انتخاب دیگر وجود نخواهد داشت.

ه) در الگوریتم simulated annealing دما شرط اصلی حلقه اصلی برنامه است و با کمک آن ما مانع گیر کردن در مینیمم محلی(یا ماکسیمم محلی)می شویم. دراثر کاهش سریع دما به سرعت داریم اینشرط را کنار میگذاریم و بعد از گیر کردن در مینیمم یا ماکسیمم محلی جلوگیری می کند را کنار میگذاریم و در نتیجه در بهینه محلی گیر خواهیم کرد.

اگر دما عددی مثبت و ثابت باشد سرعت الگوریتم SA ما به شدت کند خواهد شد چون همواره در وضعیت های بد(وضعیت هایی که دلتا منفی است) دنبال جواب خواهد بود و رفتاری مانند first choice hill climbing خواهیم داشت.

سوال ۲) الف)فضای جست و جو برابر است با !n زیرا n جایگشت داریم که برای هر کدام انتخاب میکنیم از n تا داده یکی از آن ها کم میشه و به همین ترتیب تا به پایان برسه.

ب) همسایگی به این معنی است که تنها ۲ کلمه از جمله ای که داریم را جابجا کنیم. مثلا برای جمله ی "مجازی است این ترم هوش مصنوعی" برای مثال یک همسایگی با جابجایی مجازی و مصنوعی تشکیل میشه " مصنوعی است این ترم هوش مجازی" و یا با جابجایی هوش و ترم داریم " مجازی است این هوش ترم مصنوعی " .

ج) خیر از انجایی که hill climbing روشی است که جواب optimal به ما نمیدهد پس ممکن است مسئله در بهینه محلی گیر کند البته با کمک تابع هدف میتوان این را بهینه کرد ولی باز امکانگیر کردن در بهینه محلی هست.بهینه محلی زمانی است که در جمله ای گیر کنیم که جابجایی ها بین کلمات عدد تابع هدف بهتری به ما ندهد.

د) برای کراس اور کردن از تکنیک تک نقطه کافی است دو عضو از جمعیت را انتخاب کنیم و از یک جا (مثلا محل i ام) دو کلمه parent را برش میدهیم و فرزند جدید کافی است از parent 1 تا کلمه i ام نگه داریم و از i ام به بعد به ترتیبی که در parent 2 کلمات آمده اند آن کلماتی که در نیمه اول نیستند قرار میدهیم. این کار برای این است که از تکرار کلمات جلوگیری کنیم.

مثال:

parent 1 : مجازی است این ترم هوش مصنوعی

parent 2 : مصنوعی این است ترم هوش مجازی

فرزند اول: مجازی است این مصنوعی ترم هوش

فرزند دوم: مصنوعی این است مجازی ترم هوش

گزارش پیاده سازی(الگوریتم SA)

```
class Graph:
    def __init__(self, v):
        self.v = v
        self.graph = defaultdict(list)

    def addEdge(self, s, d):
        self.graph[s].append(d)

    def parent(self, s, d):  # check if s eating d
        for v in self.graph[s]:
            if (v == d):
                return True
        return False
```

در ابتدا با class graph شروع میکنیم. این کلاس برای ساخت گراف ماست. و اطلاعات مربوط به گرافمان را در آن ذخیره می کنیم. که این اطلاعات راس ما و راس هایی که به آن ها یال دارند هست.

در تابع addEdge دو ورودی داریم که یک راس و دومی راسی است که یال بین آن ها وجود دارد و کار این تابع اضافه کردن یال برای راس s هست.

و تابع parent ورودی مشابه تابع بالا میگیرد اما این تابع برای چک کردن این است که آیا رابطه ی شکارچی بودن راس اول و شکار بودن راس دوم برقرار است ویا خیر.

```
def objective(sol):
    q = 0
    for i in range(n):
        a = sol[i]
        for j in range(i, n):
            b = sol[j]
            if (graph.parent(b, a)):
                q = q + 1
    return q
```

حال وارد الگوریتم sa می شویم:

این تابع که همان تابع هدف است برای بدست آوردن نا بجایی هاست. به این شکل که دونه دونه از خانه اول به ترتیب درایه اول را فیکس کرده و چک میکنیم که آیا شکارچی آن بعدش آمده است ویا خیر و هر جا که این اتفاق بیوفتد متغیر را ۱ واحد اضافه می کنیم.

برای خواندن از روی فایل از این کد استفاده میکنیم و هر خط آن را در یک لیست که تعریف کردیم میریزیم

```
lines = []
with open("input.txt") as f:
    lines = f.readlines()
```

```
n = int(lines[0])
lines.pop(0)
T = 100000
t_change = 0.99
```

در این قسمت متغیر هامون رو تعریف میکنیم که T همان دما است که عددی بزرگ بهش دادیم و t_change ضریب کاهش T است و n هم تعداد رئوس گراف است که همان طور که در توضیحات مسئله گفته شد خط اول فایل ورودی تعداد رئوس است و چون در ادامه به آن نیاز نداریم آن را از لیست حذف می کنیم.

```
graph = Graph(n)
for l in lines:
    e = l.split(' ')
    graph.addEdge(int(e[0]), int(e[1]))
```

حال که تعداد راس ها را داریم یک گراف با کلاس graph میسازیم و برای دادن اطلاعات گراف هر خط را میخوانیم و با توجه به جداسازی عنصر اول را به عنوان راس و عنصر دوم را به عنوان یالی خارج میشه میذاریم.

```
sequence = [i for i in range(1, n + 1)]
solution = random.sample(sequence, n)
fitness = objective(solution)
```

در این قسمت یک لیست تعریف میکنیم که عناصر ان از ۱ تا n+1 باشد که این همان رئوس ما هستند (حیوانات سوال) و حال ترتیب آن را بهم میزنیم و حال تعداد نابجایی های این لیست جدید را بدست می آوریم و آن را نگه میداریم.

```
while T > 0:
    neighbor = solution.copy()
    temp = np.random.randint(n)
    temp2 = np.random.randint(n)

    while ((graph.parent(neighbor[temp], neighbor[temp2]) and (temp < temp2))):
        temp = np.random.randint(n)
        temp2 = np.random.randint(n)

    temp_data = neighbor[temp]
    neighbor[temp] = neighbor[temp2]
    neighbor[temp2] = temp_data

    fit = objective(neighbor)

    delta = fitness - fit
    if delta >= 0:
        solution = neighbor
        fitness = fit
    else:
        pr = math.exp(delta / T)
        if pr >= .95:
            solution = neighbor
            fitness = fit

    # print(fitness)
    T = int(T * t_change)
```

در این بخش کد یک حلقه while داریم که در آن به دنبال بهتر کردن لیست و ترتیب آن هستیم. شرط تکرار حلقه هم مثبت بودت T است. یک همسایگی برای لیست اولیه پیدا میکنیم به این صورت که به صورت تصادفی دو درایه از آن را در نظر میگیریم و چک میکنیم که آیا این دو رابطه شکار و شکارچی بودن دارند که اولی شکار دومی باشد که با تغییر آن ها به سمت بهتر شدن بریم (این موضوع که جایگاه شکار بعد شکارچی باشد که با تغییر بهتر شود هم چک میکنیم) و اگر این رابطه برقرار نبود دوباره متغیر تصادفی انتخاب میکنیم تا شرط لازم رو داشته باشه. حال با رسیدن به متغیر های درست جای آن ها را عوض میکنیم و تعداد نابجایی های این همسایگی را بدست می آوریم و با لیست اصلی مقایسه میکنیم اگر کمتر بود که این همسایگی را به عنوان لیست جدید میپذیریم و اگر هم نه با حساب کردن احتمال و درصد احتمالی که داریم هم باز اگر به سمت بد شدن بود قبول میکنیم و حال T را در ضریب کاهش T ضرب کرده تا زمانی که به صفر میل کند و تکرار حلقه به پایان برسد.

و یک نمونه از خروجی کد :

```
C:\Users\ASUS\PycharmProjects\pythonProject\9712045.2\1\venv\Scripts\python.ex
[20, 2, 15, 4, 3, 1, 11, 10, 19, 18, 12, 16, 8, 7, 13, 6, 14, 9, 17, 5]
2

Process finished with exit code 0
```