گزارش پروژه 1 الگوریتم- سید حسن سعادت میرقدیم – 9631040

1- در پیاده سازی الگوریتم nearest neighbor ابتدا در فایل utils.py کلاس Point را به عنوان نقطه تعریف کردیم که مشخصه های طول و عرض (x, y) نقطه را در خود ذخیره میکند و همچنین مشخصه ای به عنوان visited دارد که از نوع بولین بوده و true بودن آن مشخص میکند که از آن نقطه عبور شده است. متد dist (در فایل utils.py)نیز با گرفتن دو نقطه ی p1 , p2 فاصله ی آن دو از هم را محاسبه کرده و بازمیگرداند. متد get_point_list نیز داده ها را به صورت ذکر شده در صورت سوال از کاربر میگیرد. متد has_unvisited لیستی از نقاط را گرفته بر روی آن نقاط پیمایش کرده و اگر نقطه یا نقاطی وجود داشت که از آنها عبور نشده بود مقدار true برمیگرداند. متد generate_dist_matrix با گرفتن لیست نقاط ماتریسی n*n تشکیل میدهد که فاصله ی بین هر دو نقطه در آن مشخص میشود در واقع از دو حلقه ی for (به صورت تو در تو) تشکیل شده که در داخل آن فاصله ی هر دو نقطه با استفاده از متد distمحاسبه شده و در درایه ی مناسب قرار داده میشود که پیچیدگی آن از مرتبه ی $O(n^2)$ است.

در نهایت متد nnm_shortest_path_calculate با گرفتن لیست نقاط ورودی به ترتیب زیر جواب را محاسبه می کند:
پس از اجرای برنامه ابتدا متد get_point_list تعداد و خود نقاط را از کاربر میگیرد سپس آنها را به متد nnm_shortest_path_calculate پاس میدهد و این متد در ابتدا با دادن لیست نقاط مذکور به متد generate_dist_matrix ماتریس فاصله ی نقاط را میگیرد و در dist_matrix ذخیره میکند، سپس نقطه ی اول لیست را به عنوان مبدا در نظر گرفته(طبق فرض سوال) در متغیر tmp ذخیره کرده و مشخصه ی visited آن را true قرار میدهد و آن را به لیست نقاط(متغیر path) اضافه میکند،سپس حلقه ی while با شرط اینکه هنوز نقاطی برای عبور وجود دارد اجرا میشود لازم به ذکر است پیچیدگی محاسبه ی شرط این حلقه از مرتبه ی O(n) میباشد.
سپس در درون حلقه مقداربزرگی را به عنوان مینیمم فاصله ی دو نقطه در نظر میگیریم(min_dist=10000) و متغیر کمکی tmp1 را نیز تعریف میکنیم، سپس در حلقه ی for که پیمایشی بر روی لیست نقاط است هر بار فاصله ی هر نقطه p تا نقطه ی tmp- که در ابتداهمان نقطه اول لیست است- را از طریق ماتریس dist_matrix گرفته و اگر از آن نقطه عبور نشده بود و فاصله آن تاtmp کمتر از min_dist بود متغیر min_dist آپدیت شده و مقدارtmp1 برابر با نقطه ی p میشود و مشخصا تا پایان اجرای for نقطه ی با کمترین فاصله مشخص شده و در متغیر tmp قرار میگیرد و مشخصه ی visited آن true میشود و به لیست خروجی اضافه میشود(متغیر path) در پایان نیز اولین نقطه که همان آخرین نقطه ی عبوریست به لیست خروجی اضافه شده و در نهایت بر روی صفحه لیست نقاط عبوری پرینت میشود. پیچیدگی این الگوریتم نیز میشود $O(n^2 + n*n) = O(n^2)$.

در الگوریتم exhaustive search نیز از متد ها و کلاس های کمکی در فایل utils.py استفاده شده که در بالا شرح عملکرد هر کدام به تفصیل ذکر گشته است. اما بعد:
متد get_points_permutations با گرفتن لیست نقاط و جایگشت های ممکنه برای اعداد 1 تا n-1 (nتعداد نقاط است)که همان جایگشت اندیس های لیست نقاط است(بدون احتساب نقطه ی اول چون آن را همیشه مبدا در نظر میگیریم) ، جایگشت های ممکن برای نقاط لیست را حساب میکند و در لیستی قرار داده و برمیگرداند. پیچیدگی این عملیات O(n) است. متد get_path_list با گرفتن لیست نقاط، لیست جایگشت اعداد 1تا n-1 را در متغیر perms قرار میدهد سپس در حلقه ی for به ازای هر جایگشت در perms، متغیر perm_list تکمیل میشود و به path_list اضافه میشود در واقع تکمیل شدنش به این معنی است که نقطه ی مبدا به ابتدا و انتهای آن اضافه شده و جایگشت نقاط دیگر به وسط آن اضافه

میشود با استفاده از متد get_points_permutations. سپس در حلقه ی for دیگری هربار طول مسیر محاسبه شده در قبل به انتهای آن مسیر اضافه میشود و سپس لیست مسیر بازگردانده میشود.
در متد esm_calculate_shortest_path نیز مینیمم طول هر مسیر از مقایسه طول تمام مسیر محاسبه میشود و در نهایت کم طول ترین مسیر بازگردانده میشود.
در نهایت نیز هم مسیر ها و هم کم طول ترین مسیر چاپ میشود.
پیچیدگی این الگوریتم در نهایت میشود $O(n! + n*n + n*n + n) = O(n!)$

2- پاسخ این سوال در جواب بالا مشخص شد، برای اولین الگوریتم این مقدار برابر $O(n^2)$ و برای دومین برابر با $O(n!)$ میباشد.

3-

اعداد رندوم در بازه ی -20 تا 20 در نظر گرفته شده اند.

algorithm	n	Run 1	Run 2	Run 3	average
Nearest neighbor	100, 200, 400, 500	0.0644604 0.3426857 2.1944908 3.9023387	0.0573808 0.3554774 2.1129995 4.147751	0.060460 0.3699417 2.1399866 3.7627192	0.0607670 0.3560349 2.1491589 3.937603
Exhaustive search	7, 8, 9, 10	0.0125155 0.1179986 0.9630447 9.345388	0.0129932 0.1102096 1.0302206 9.6250428	0.0155213 0.1214162 0.9635166 9.2694038	0.0136766 0.1165415 0.9855939 9.4132782

4-

با توجه به اعداد درون جدول در exhaustive search میبینیم که با افزایش یکی یکی عدد n هر بار زمان اجرا در عدد بعدی n ضرب میشود مثلا وقتی n=9 است زمان آن برابر 0.9630447 است و زمانی که n=10 میباشد زمان اجرا 9.345388 است و تقریبا 10 برابر زمان قبلی بنابراین میتوان نتیجه گرفت که واقعا پیچیدگی زمانی این الگوریتم از مرتبه !n است و همینطور نیز برای الگوریتم nearest neighbor نیز از مقایسه اعداد میتوان نتیجه گرفت که پیچیدگی زمانی این الگوریتم برابر $O(n^2)$ است.