علیرضا روحانی-9531035

گزارش پروژه 1 طراحی الگوریتم

مقدمه: در ابتدا قرار است ورودیهای مد نظر(مختصات شهر ها) را بگیریم و همچنین دیتا استراکچر های مربوطه را تعیین کنیم :

ابتدا تعداد شهر ها را به صورت n میگیریم. آرایه tour را برای مسیر مورد نظر تعیین میکنیم که ترتیب شهرها قرار دارد و عدد tourLenght برای طول مسیر، هم چنین مختص طولی هر شهر را در ارایه longs ذخیره میکنیم و عرضی را در آرایه lats.

```
int n = scanner.nextInt();

ArrayList<Integer> tour = new ArrayList<>();
double tourLength = 0;

ArrayList<Double> longs = new ArrayList<>();
ArrayList<Double> lats = new ArrayList<>();
```

O(1)

.2 & 1

حال به شرح هرکدام از الگوریتمها میپردازیم:

الگوریتم nearest-neighbour:

ابتدا برای تعیین ویزیت شدن هر شهر یک آرایه بولین درنظر میگیریم. سپس همه المنت های آنرا معادل false میگذاریم زیرا در حالت اولیه هیچکدام از شهرها ویزیت نشده اند. سپس شهر اول را به tour ادد میکنیم زیرا تفاوتی در اجرای الگوریتم ایجاد نمیکند و برای همگام شدن با سایر الگوریتمها آنرا از شهر اول شروع میکنیم. پس مقدار بولین visited آن شهر را برابر true قرار میدهیم زیرا از روی آن عبور کردیم.

متغیر minDistance را برای ذخیره مسیر تا نزدیکترین شهر تولید میکنیم و مقدار آنرا بصورت پیشفرض معادل بزرگترین عدد double قرار میدهیم که در مقایسه اعداد بزرگ دچار مشکل نشویم.

```
ArrayList<Boolean> visited = new ArrayList<>();

for (int i = 0; i < n; i++) {
   longs.add(scanner.nextDouble());
   lats.add(scanner.nextDouble());
   visited.add(false);
}

tour.add(0);
visited.set(0, true);
double minDistance = Double.MAX_VALUE;
```

O(1)

حال دو متغیر current و next را برای ذخیره شهری که روی آن قرار داریم و شهر بعدی که قرار است به آن سفر کنیم، تعریف میکنیم.

```
int current = 0;
int next = 0;
```

O(1)

حال با توجه به الگوریتم پیش میرویم و از شهر اول شروع کرده و سپس به نزدیکترین شهر رفته، به این صورت فاصله این شهر را با تمام شهرها حساب میکنیم (با استفاده از تابع measureDistance) و کوتاهترین فاصله مربوط به نزدیکترین شهر است. سپس از آن شهر نیز اینکار را تکرار میکنیم تا شهر ها تمام شود. نکته ای که باید درنظر گرفته شود این است که پس از عبور از هر شهر باید مقدار visited آن شهر را برابر true قرار دهیم و در پیمایش هر شهر به سراغ شهرهایی برویم که هنوز ویزیت نشده اند و مقدار

visited آنها برابر false میباشد. سپس فاصله سفر تاهر شهر را به مقدار tourLenght اضافه میکنیم. همچنین پس از عبور از هر شهر مقدار current را برابر آخرین شهری که ویزیت شده قرار میدهیم و مقدار minDistance را دوباره برابر بزرگترین مقدار double قرار میدهیم تا در محاسبه فاصله شهر فعلی با شهرهای دیگر دچار مشکل نشویم.

تابعی که فاصله شهرها را حساب میکند به صورت زیر است.(مختصات دو شهر را گرفته و فاصله آنهارا به صورت خروجی به ما میدهد)

```
private static double measureDistance(double x1, double y1, double x2, double y2) {
  return Math.pow((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2), 0.5);
}
```

O(1)

همچنین پیاده سازی روش ذکر شده نیز به صورت زیر میباشد.

```
while (tour.size() < n) {
  for (int i = 0; i < n; i++) {
    if (!visited.get(i) && measureDistance(longs.get(i), lats.get(i),
longs.get(current), lats.get(current)) < minDistance) {
      minDistance = measureDistance(longs.get(i), lats.get(i),
longs.get(current), lats.get(current));
      next = i;
    }
  }
  tour.add(next);
  tourLength = tourLength + measureDistance(longs.get(current),
lats.get(current), longs.get(next), lats.get(next));
  visited.set(next, true);
  current = next;
  minDistance = Double.MAX_VALUE;
}
```

O(n×n)

در انتها کافیست شهر اول را نیز به tour اضافه کنیم تا تورمان کامل شود و به شهر اول برگردد. همچنین فاصله اخرین شهر تا شهر اول را نیز به tourLenght اضافه میکنیم.

در انتها نیز کافیست مختصات شهرهای مسیر(با توجه به آرایه tour) را به ترتیب چاپ کنیم.

```
tour.add(0);
tourLength = tourLength + measureDistance(longs.get(current),
lats.get(current), longs.get(0), lats.get(0));

System.out.println("Nearest-neighbour ");
System.out.println("The order in which the points are visited: ");
for (int i = 0; i < tour.size(); i++) {
   System.out.println((i + 1) + ". (" + longs.get(tour.get(i)) + "," +
lats.get(tour.get(i)) + ")");
}
System.out.println("Tour length: " + tourLength);
System.out.println();
```

O(n)

الگوریتم exhustive:

ابتدا آرایه tour را تخلیه میکنیم.

چون در این الگوریتم قرار است پیمایش های مختلف را مقایسه کنیم دو متغیر یکی برای ذخیره طول پیمایش انجام شده و یکی برای ذخیره کوتاهترین طول موجود میان پیمایش ها درنظر میگریم.

همچنین متغیر دیگر sol را میسازیم که توسط آن پیمایش مد نظر را(میان سایر پیمایش ها) انتخاب کنیم.

سپس یک آرایه با سایز n میسازیم که شماره شهرها (از صفرتا n-1) را به آن میدهیم که بتوانیم جایگشت های مختلف آنرا حساب کنیم. (دقت کنید که شماره شهرها بر اساس ترتیبی که به ورودی داده میشوند مشخص میشود)

```
tour.clear();
double tourDistance = 0;
double minTourDistance = Double.MAX_VALUE;
int sol = -1;
ArrayList<Integer> allCities = new ArrayList<>();
for (int i = 0; i < n; i++) allCities.add(i);
```

O(n)

در این جا نیاز به پیدا کردن جایگشت های مختلف آرایه allCities داریم که برای آن یک کلاس به اسم Permute طراحی شده است.

کلاس Permute شامل یک متد به نام permute میباشد که بعنوان ورودی یک arraylist از اعداد integer میگیرد و به ما تمامی جایشگت های آن اعداد را میدهد و آنرا در فیلد استاتیک کلاس ذخیره میکند. فیلد ما یک arraylist از آرایه هاست که تمامی جایشگتهای مد نظر هر کدام به صورت یک آرایه در arraylist ذخیره میشود. (با این حساب که به خانه صفرم چون شهر اول است کاری نداشته و جایگشت سایر خانه ها یعنی از 1 به بعد را محاسبه میکند و در فیلد میریزد)

```
public class Permute {

   static ArrayList<int[]> permutations = new ArrayList<>();

   public static void permute(ArrayList<Integer> arr) {
      permuteHelper(arr, 1);
   }

   private static void permuteHelper(ArrayList<Integer> arr, int index) {

      if (index >= arr.size() - 1) {
```

```
            int[] temp = new int[arr.size()];
            for (int i = 0; i < arr.size() - 1; i++) {
                temp[i] = arr.get(i);
            }
            temp[arr.size() - 1] = arr.get(arr.size() - 1);
            permutations.add(temp);
            return;
        }

        for (int i = index; i < arr.size(); i++) {

            int t = arr.get(index);
            arr.set(index, arr.get(i));
            arr.set(i, t);

            permuteHelper(arr, index + 1);

            t = arr.get(index);
            arr.set(index, arr.get(i));
            arr.set(i, t);
        }
    }
}
```

O((n-1)!)

دلیل !(n-1) این است که index از 1 شروع شده. n-1 بار تابع صدا میشود و تابع دارای حلقه حداکثر n تایی است.

حال آرایه allCities را permute میکنیم و تمام جایشگت های آنرا در متغیر permutation میریزیم که یک arraylist از آرایه هاست.

```
Permute.permute(allCities);
ArrayList<int[]> permutations = Permute.permutations;
```

O((n-1)!)

دلیل این مرتبه این است که از تابع permute استفاده شده است.

حال تمام جایگشت ها را داریم. کافیست آنهارا پیمایش کنیم و کوتاهترین مسیر را بیابیم.
روی سفرها ها حلقه میزنیم و طول هر سفر را محاسبه میکنیم و در نهایت کوتاهترین سفر را پیدا میکنیم.

```
int t;

for (int i = 0; i < permutations.size(); i++) {
   for (int j = 0; j < permutations.get(i).length - 1; j++) {
      tourDistance += measureDistance(longs.get(permutations.get(i)[j]),
lats.get(permutations.get(i)[j]), longs.get(permutations.get(i)[j + 1]),
lats.get(permutations.get(i)[j + 1]));
   }
   t = permutations.get(i).length - 1;
   tourDistance += measureDistance(longs.get(permutations.get(i)[t]),
lats.get(permutations.get(i)[t]), longs.get(permutations.get(i)[0]),
lats.get(permutations.get(i)[0]));
   if (tourDistance < minTourDistance) {
      minTourDistance = tourDistance;
      sol = i;
   }
   tourDistance = 0;
}
```

O(n×(n-1)!)=O(n!)

حال کوتاهترین مسیر را پیدا کردیم و آنرا معادل tourLenght میگذاریم. و همچنین شهرهای آن مسیر را به tour ادد میکنیم. و درنهایت شهر اول را نیز به tour ادد میکنیم تا دور کامل شود. سپس آنهارا مانند حالت قبل چاپ میکنیم.

```
tourLength = minTourDistance;
for (int k = 0; k < permutations.get(sol).length; k++) {
   tour.add(permutations.get(sol)[k]);
}
tour.add(permutations.get(sol)[0]);
System.out.println("Exhustive ");
System.out.println("The order in which the points are visited: ");
for (int i = 0; i < tour.size(); i++) {
   System.out.println((i + 1) + ". (" + longs.get(tour.get(i)) + "," +
lats.get(tour.get(i)) + ")");
}
System.out.println("Tour length: " + tourLength);
System.out.println();
```

O(n)

حال با توجه به مرتبه های بدست آمده متوجه میشویم که الگوریتم nearest-neighbor دارای مرتبه O(n^2) میباشد. و همچنین الگوریتم exhustive دارای مرتبه O(n!) میباشد.

3. ابتدا یک random number generator میسازیم. بصورتی که مختصات رندوم برای شهرها بدهد.

```
Random random = new Random();
for (int i = 0; i < n; i++) {
   longs.add(random.nextDouble() * 100);
   lats.add(random.nextDouble() * 100);
   visited.add(false);
}
```

O(n)

حال به انتخاب n های مختلف میپردازیم. برای الگوریتم exhustive باید n های کوچکتر از 13 انتخاب کنیم زیرا به مشکل memory برمیخوریم (تست شده) پس نمیتوان !13 را تست نمود زیرا از لحاظ حافظه به مشکل میخوریم. پس n های پیشنهادی برای الگوریتم exhustive باید زیر 13 باشد. و ما اعداد 12 و 11 و 10 و 9 را در نظر میگیریم.

برای الگوریتم nearest-neighbour چون مرتبه آن از n^2 میباشد، دستمان بازتر است و میتوانیم n های بزرگتری انتخاب کنیم که دقت تست برای ما بالاتر باشد. n های پیشنهادی برای این الگوریتم اعداد 50 و 500 و 5000 و 50000 میباشد.

حال شرح تست داده ها به صورت جدول زیر است.(واحد ها بر حسب ثانیه میباشد)

ابتدا برای الگوریتم exhustive را محاسبه میکنیم:

n/runtime	Runtime1	Runtime2	Runtime3	Avg runtime
9	0.099	0.088	0.094	0.093
10	0.36	0.31	0.39	0.35
11	2.06	2.02	2.04	2.04
12	22.01	21.88	21.73	21.87

برای الگوریتم nearest-neighbor نیز به شرح زیر است:

n/runtime	Runtime1	Runtime2	Runtime3	Avg runtime
50	0.0020	0.0019	0.0021	0.0020
500	0.026	0.025	0.028	0.0263
5000	0.49	0.57	0.50	0.52
50000	49.3	48.5	50.8	49.53

4. حال کافیست مشاهدات تجربی را با تئوری مقایسه کنیم. درحالت تئوری برای الگوریتم exhustive توقعمان این است که بصورت فاکتوریلی زمان اجرا زیاد شود. یعنی از لحاظ

تئوری میخواهیم که مثلا با تغییر n از 10 به 11 زمان اجرا 11 برابر شود. حال مشاهدات را با مبحث تئوری مقایسه میکنیم.

با تغییر n از 9 به 10 زمان اجرا 3.76 برابر شده است درحالیکه توقع داشتیم 10 برابر شود.(این بخاطر لود و محاسبات ثابت سیستمی است که در زمان اجرا تاثیر گذاشته است)

با تغییر n از 10 به 11 زمان اجرا 5.82 برابر شده است درحالیکه توقع داشتیم 11 برابر شود.(این نیز به خاطر لود سیستم است اما میبینیم که به نتیجه مد نظر نزدیکتر شده ایم)

با تغییر n از 11 به 12 زمان اجرا 10.72 برابر شده است درحالیکه توقع داشتیم 12 برابر شود.(میبینیم که بسیار به نتیجه مطلوب نزدیک شده ایم زیرا محاسبات کوچک با مرتبه کمتر کم اثر میشود)

پس درمیابیم که در بینهایت به نتیجه تئوری خواهیم رسید و آزمایش درست بوده است.

حال به الگوریتم nearest-neighbor میپردازیم. درحالت تئوری برای این الگوریتم توقعمان این است که با مرتبه n^2 زمان اجرا زیاد شود. یعنی اگر n مثلا 10 برابر شد زمان اجرا 10^2 یعنی 100 برابر شود. حال مبحث نظری را با مشاهدات تجربی مان مقایسه میکنیم.

با تغییر n از 50 به 500 زمان اجرا 13.5 برابر شده است درحالیکه نتیجه مد نظر 100 برابر میباشد.(این موضوع بخاطر لود و محاسبات سیستم میباشد)

با تغییر n از 500 به 5000 زمان اجرا 19.7 برابر شده است درحالیکه نتیجه مد نظر 100 برابر میباشد.(این نیز بخاطر لود و محاسبات سیستم میباشد)

با تغییر n از 5000 به 50000 زمان اجرا 95.2 برابر شده است درحالیکه نتیجه مد نظر 100 برابر میباشد.(بسیار به نتیجه مطلوب نزدیک شده ایم زیرا محاسبات کوچک که مرتبه کمتری دارند کم اثر شده اند)

پس باز هم درمیابیم که در بینهایت به نتیجه تئوری خواهیم رسید و آزمایش درست بوده است و در بینهایت دارای مرتبه n^2 میباشد.