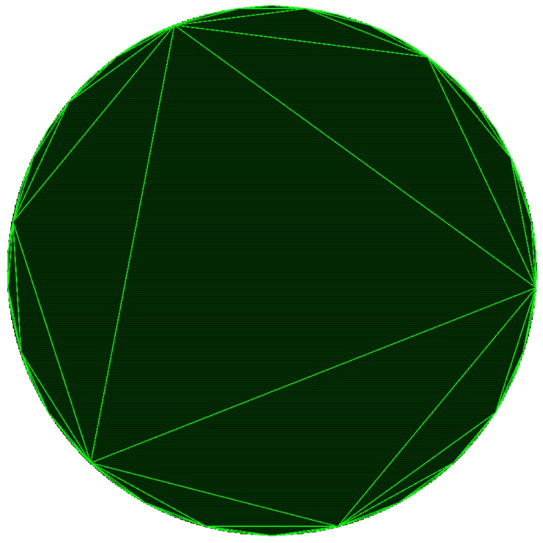
COMP361

Assignment 2



Dynamic Programming

William Kilty

300473541

Question 1:

```
Triangulate(n, verts) {
    lengths[n][n] <- 0 //initialise everything to zero
    for (diag = 2; < n; ++) {
        for (j = diag; < n; ++) {
            i <- j-diag
            lengths[i][j] <- Infinity
            for (k = i+1; < j; ++) {
                perim <- distance(verts[i], verts[j])
                    + distance(verts[j], verts[k])
                    + distance(verts[k], verts[i])
                lengths[i][j] <- min(
                    lengths[i][j],
                    lengths[i][k] + lengths[k][j] + perim
                )
            }
        }
    }
    return lengths[0][n-1]
}
```

Theorem:

This algorithm displays optimal substructure.

Proof:

Assume we have optimal solution S (minimum length of triangulation).
We have sub problem solutions $S_1 = lengths[i][k]$ and $S_2 = lengths[k][j]$.

Assume S_1 not optimal:

i.e. There is some k such that
$lengths[i][k] + lengths[k][j] + perim < S_1$.

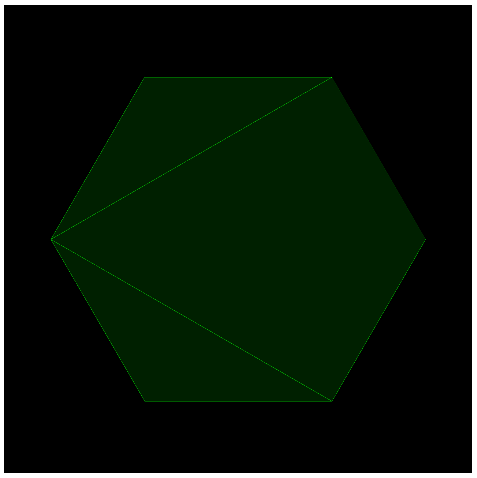
However, for each k we assign the value to S_1 iff
$lengths[i][k] + lengths[k][j] + perim$ is the minimum value.
So S_1 must by definition have the lowest k result.

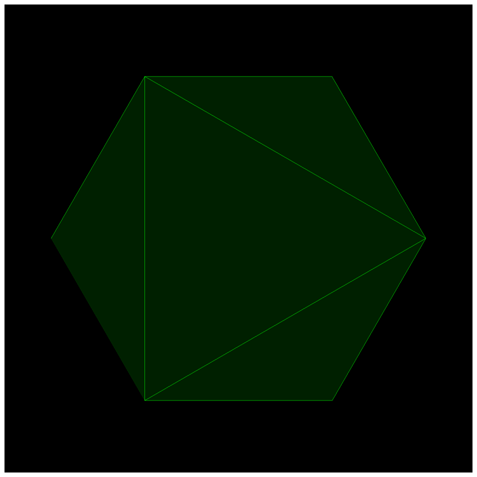
Contradiction. Therefore sub-problems must be optimal

This algorithm has a time complexity of $O(n^3)$ due to three layers of for-loops and a space complexity of $O(n^2)$ due to an array of length nxn to store the results.

Hexagon

0	0	1.49	3.39	5.06	6.56
0	0	0	1.49	3.39	5.06
0	0	0	0	1.49	3.39
0	0	0	0	0	1.49
0	0	0	0	0	0
0	0	0	0	0	0



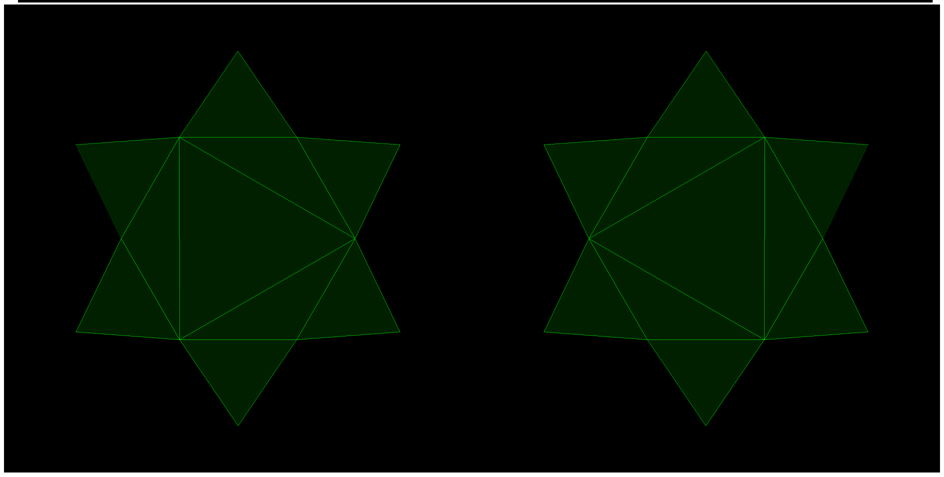


Question 2:

The algorithm I made for question 1 applies to non-convex polygons. As such, the correctness has already been proven, and time complexity is the same.

Hexagram

0	0	0.69	1.64	2.32	3.61	4.20	5.49	5.94	7.07	7.57	8.26
0	0	0	0.84	1.64	3.02	3.61	5.02	5.49	7.04	7.07	8.16
0	0	0	0	0.69	1.64	2.32	3.61	4.20	5.49	5.94	7.07
0	0	0	0	0	0.84	1.64	3.02	3.61	5.02	5.49	7.04
0	0	0	0	0	0	0.69	1.64	2.32	3.61	4.20	5.49
0	0	0	0	0	0	0	0.84	1.64	3.02	3.61	5.02
0	0	0	0	0	0	0	0	0.69	1.64	2.32	3.61
0	0	0	0	0	0	0	0	0	0.84	1.64	3.02
0	0	0	0	0	0	0	0	0	0	0.69	1.64
0	0	0	0	0	0	0	0	0	0	0	0.84
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0



Question 3:

The KMP string search algorithm is indeed a dynamic programming problem.

KMP string search works by keeping a lookup array of chars in a pattern, and comparing it to a given string. If there's a mismatch, the algorithm shifts the pattern forwards to the next potential match in such a way that requires no backtracking or repeating of past work.

This ability to reuse past checks in order to not repeat past searches is in line with the process of dynamic programming.

We run on the assumption of optimal substructure and repeatable subproblems to skip over unnecessary data and end up with an algorithm of O(n) complexity. This is a fundamental requirement for dynamic programming, and therefore allows one to classify KMP string search as a dynamic programming algorithm.

Question 4:

```
Gerrymander(m, precincts) {
        n <- precincts.length
        S[n][n/2][mn][mn] <- false //initialise everything to false
        S[0][0][0][0] <- true //initialise first index to true
        for (j = 0; < n; ++) {
                for (k = 0; < min(j, n/2); ++) {
                        for (x = 0; < jm; ++) {
                                for (y = 0; < jm; ++) {
                                        S[j][k][x][y]
                                                 = S[j-1][k][x-precincts[j]][y]
                                                || S[j-1][k-1][x][y-precincts[j]]
                                }
                        }
                }
        }
        // find answer
        for (x = mn/4; < mn; ++) {
                for (y = mn/4; < mn; ++) {
                        // if any point in this region true, then gerrymandering possible
                        if (arranges[n][n/2][x][y]) return true
                }
        }
}
```

Theorem:

This algorithm displays optimal substructure.

Proof:

Assume we have optimal solution S (whether a given set of districts always wins).

We have sub problem solutions

$S_1 = S[j - 1][k][x - precincts[j]][y]$ and $S_2 = S[j - 1][k - 1][x][y - precincts[j]]$.

Assume S_1 not optimal:

i.e. There is some value such that

`S[j-1][k][x-precincts[j]][y] || S[j-1][k-1][x][y-precincts[j]]` != S_1.

However, we assign the value to S_1 iff

$S[j - 1][k][x - precincts[j]][y] \,||\, S[j - 1][k - 1][x][y - precincts[j]]$ holds.

So S_1 must by definition have the correct result.

Contradiction. Therefore sub-problems must be optimal

This algorithm has a time complexity of $O(n^4)$ due to four layers of for-loops and a space complexity of $O(n^4)$ due to an array of length nx(n/2)x(mn)x(mn) to store the results.