

- CODE FOR n=5 ATTEMPT (WITHOUT CHATGPT)

```
/* COP 3502C Assignment 3
This program is written by: Lucas Luna */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

//Creates a struct for the garages
typedef struct garage {
    char name[20];
    int x;
    int y;
} garage;

//Prints out the array
void print(int *A, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");
}

//Calculates the distance between two points
float distance(int x1, int y1, int x2, int y2){
    float distance = sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
    return distance;
}

//Calulates the the shortest distances between all the garages
float calculateSum(int* perm, garage* garages, int n, float* bestVal, int* bestPerm){
    float sum = 0;
    for(int i = 0; i < n; i = i+2){
        sum += distance(garages[perm[i]].x, garages[perm[i]].y, garages[perm[i+1]].x,
garages[perm[i+1]].y);
    }

    if(sum < *bestVal){
        *bestVal = sum;
        for(int i= 0; i < n; i++){
            bestPerm[i] = perm[i];
        }
    }
}
```

```

}

return sum;
}

//Permutation function
void perms(int* perm, garage* garages, int* used, int k, int n, float* bestVal, int* bestPerm) {
    if (k == n) {
        calculateSum(perm, garages, n, bestVal, bestPerm);
        return;
    }
    for (int i = 0; i < n; i++) {
        if (!used[i]){
            used[i] = 1;
            perm[k] = i;
            perms(perm, garages, used, k + 1, n, bestVal, bestPerm);
            used[i] = 0;
        }
    }
}

int main(void) {
    //Scans the number of expressways
    int expresswayCnt;
    scanf("%d", &expresswayCnt);

    //
    int garageCnt = expresswayCnt * 2;
    float bestValue = 999;

    //Initializes the perm array
    int perm[garageCnt];
    for (int i = 0; i < garageCnt; i++) {
        perm[i] = 0;
    }

    //Initializes the used array
    int used[garageCnt];
    for (int i = 0; i < garageCnt; i++) {
        used[i] = 0;
    }

    //Initializes the bestPerm array
}

```

```

int bestPerm[garageCnt];
for (int i = 0; i < garageCnt; i++) {
    bestPerm[i] = 0;
}

//Allocates memory for the garages
garage* garages = malloc(sizeof(garage) * garageCnt);

//Scans for the x and y coordinates and the names of the garages
for (int i = 0; i < garageCnt; i++){
    scanf("%d %d", &garages[i].x, &garages[i].y);
    scanf("%s", garages[i].name);
}

//Calls the permutation function
perms(perm, garages, used, 0, garageCnt, &bestValue, bestPerm);

//Prints out the calculated the shortest distance with the distances of the garages that were used
printf("%.3f\n", bestValue);
for(int i = 0; i < garageCnt; i = i+2){
    printf("(%s, ", garages[bestPerm[i]].name);
    printf("%s, ", garages[bestPerm[i+1]].name);

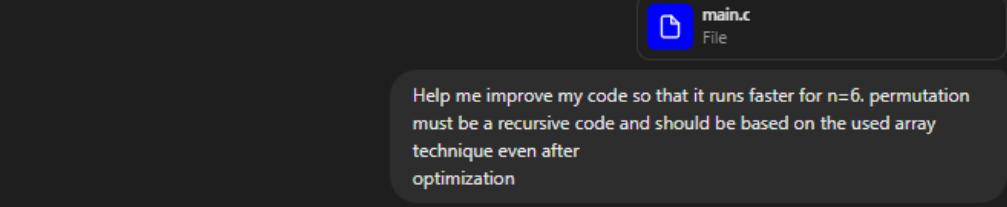
    printf("%.3f)\n", distance(garages[bestPerm[i]].x, garages[bestPerm[i]].y,
garages[bestPerm[i+1]].x, garages[bestPerm[i+1]].y));
}

//Frees allocated memory
free(garages);

return 0;
}

```

- SCREENSHOTS OF CHATGPT PROMPTS AND RESPONSES



The uploaded code is using a recursive permutation approach with a `used` array, and the goal is to optimize the code while keeping this recursive structure. For `n=6`, the current permutation generation works but can be slow due to redundant distance calculations and lack of pruning during the recursion.

Here are a few optimization ideas to improve the performance:

- 1. Cache distances between garages:**
Instead of recalculating distances repeatedly, we can cache the distances between each pair of garages. This can significantly reduce the number of floating-point operations.
- 2. Prune permutations early:**
When generating permutations, if the current partial distance sum exceeds the best distance found so far, we can terminate that branch early (pruning). This avoids unnecessary deeper recursion when a path is already worse than the best found so far.
- 3. Reduce floating-point precision errors:**
Reducing the number of times floating-point arithmetic is used can improve performance slightly, especially if the calculations are done repeatedly within the recursive calls.

Here's an optimized version of your code that implements these changes while maintaining the recursive nature with the `used` array:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef struct garage {
    char name[20];
    int x, y;
} garage;

// Global distance cache
float **distanceCache;

// Function to precompute distances between all garages
void computeDistanceCache(garage* garages, int n) {
    distanceCache = (float**)malloc(n * sizeof(float*));
    for (int i = 0; i < n; i++) {
        distanceCache[i] = (float*)malloc(n * sizeof(float));
        for (int j = 0; j < n; j++) {
            distanceCache[i][j] = sqrt(pow(garages[j].x - garages[i].x, 2) + pow(garages[j].y - garages[i].y, 2));
        }
    }
}
```

```

// Function to calculate the sum of distances for a given permutation
float calculateSum(int* perm, int n) {
    float sum = 0;
    for (int i = 0; i < n; i += 2) {
        sum += distanceCache[perm[i]][perm[i+1]];
    }
    return sum;
}

// Recursive permutation function with early pruning
void perms(int* perm, garage* garages, int* used, int k, int n, float* bestVal, int* bestP)
{
    // If we've formed a complete permutation, check its sum
    if (k == n) {
        if (currentSum < *bestVal) {
            *bestVal = currentSum;
            for (int i = 0; i < n; i++) {
                bestPerm[i] = perm[i];
            }
        }
        return;
    }

    // Try all unused elements
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            used[i] = 1;
            perm[k] = i;

            // Calculate partial sum if it's a pair (even index)
            float newSum = currentSum;
            if (k % 2 == 1) {
                newSum += distanceCache[perm[k-1]][perm[k]];
            }

            // Early pruning if the current sum exceeds the best value
            if (newSum < *bestVal) {
                perms(perm, garages, used, k + 1, n, bestVal, bestPerm, newSum);
            }

            // Backtrack
            used[i] = 0;
        }
    }
}

int main(void) {
    int expresswayCnt;
    scanf("%d", &expresswayCnt);

    int garageCnt = expresswayCnt * 2;
    garage* garages = malloc(sizeof(garage) * garageCnt);

    // Input the garages' coordinates and names
    for (int i = 0; i < garageCnt; i++) {
        scanf("%d %d %s", &garages[i].x, &garages[i].y, garages[i].name);
    }

    // Precompute the distances between all garages
    computeDistanceCache(garages, garageCnt);

    int perm[garageCnt];
    int used[garageCnt];
    int bestPerm[garageCnt];
    float bestValue = 9999999; // Initialize with a large value
}

```

```

// Initialize the used array
for (int i = 0; i < garageCnt; i++) {
    used[i] = 0;
}

// Call the recursive permutation function
perms(perms, garages, used, 0, garageCnt, &bestValue, bestPerm, 0.0);

// Output the best solution
printf("%.3f\n", bestValue);
for (int i = 0; i < garageCnt; i += 2) {
    printf("(%.3f, %.3f) -> %.3f\n", garages[bestPerm[i]].name, garages[bestPerm[i+1]].name,
           distanceCache[bestPerm[i]][bestPerm[i+1]]);
}

// Free the distance cache and garages memory
for (int i = 0; i < garageCnt; i++) {
    free(distanceCache[i]);
}
free(distanceCache);
free(garages);

return 0;
}

```

[Copy code](#)

Key Optimizations:

1. **Distance Cache:** Precomputes all distances between garages and stores them in a 2D array `distanceCache`, so we don't have to recalculate distances during each recursive call.
2. **Early Pruning:** When constructing a permutation, it keeps track of the current partial sum of distances. If this partial sum exceeds the best value found so far, it stops further recursion (pruning).
3. **Pass Current Sum in Recursion:** By passing the current sum along with the recursive calls, we avoid recalculating the sum from scratch at each recursive level, improving the efficiency.

Why This Will Improve Performance for $n = 6$:

- **Cached distances** reduce redundant distance calculations.
- **Pruning** drastically cuts down the number of permutations that are fully evaluated, as many branches can be discarded early if they exceed the best known solution.
- **Keeping the recursive nature** with the `used` array technique, this optimized version should provide better performance while maintaining the recursive logic structure.

Let me know how it performs or if you need any further adjustments! [\[-\]](#)

You are allowed to use only two global variables. No other global variables are allowed:

- a. One for array of garages
- b. One for the final set of permutation int array that stores the index of the garages that gave you the minimum distance



To meet the constraints of only using two global variables—one for the array of garages and one for the final set of permutations that store the index of garages yielding the minimum distance—I'll modify the previous code accordingly.

Revised Approach:

1. Global Variables:

- An array to hold the garages data (coordinates and names).
- An integer array for bestPerm to store the permutation that yields the minimum distance.

2. Local Variables: All other variables, including used, perm, and distance cache, will be kept as local variables in the functions.

Optimized Code with the Constraint:

```
c                                     □ Copy code

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Global variables as per the requirement
garage* garages;      // Global array for garages
int* bestPerm;         // Global array for the best permutation (indices)

// Define garage structure
typedef struct garage {
    char name[20];
    int x, y;
} garage;

// Function to calculate distance between two garages
float distance(int x1, int y1, int x2, int y2) {
    return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
}

// Function to calculate the sum of distances for a permutation
float calculateSum(int* perm, int n) {
    float sum = 0;
    for (int i = 0; i < n; i += 2) {
        sum += distance(garages[perm[i]].x, garages[perm[i]].y, garages[perm[i+1]].x, garages[perm[i+1]].y);
    }
    return sum;
}

// Recursive permutation function with early pruning
void perms(int* perm, int* used, int k, int n, float* bestVal, float currentSum) {
    // If we've formed a complete permutation, check its sum
    if (k == n) {
        float totalSum = currentSum;
        if (totalSum < *bestVal) {
            *bestVal = totalSum;
            for (int i = 0; i < n; i++) {
                bestPerm[i] = perm[i];
            }
        }
        return;
    }

    // Try all unused elements
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            used[i] = 1;
            perm[k] = i;

            // Calculate partial sum if it's a pair (even index)
            float newSum = currentSum;
            if (k % 2 == 1) {
                newSum += distance(garages[perm[k-1]].x, garages[perm[k-1]].y, garages[perm[k]].x, garages[perm[k]].y);
            }
        }
    }
}
```

```

        // Early pruning if the current sum exceeds the best value
        if (newSum < *bestVal) {
            perms(perm, used, k + 1, n, bestVal, newSum);
        }

        // Backtrack
        used[i] = 0;
    }
}

int main(void) {
    int expresswayCnt;
    scanf("%d", &expresswayCnt);

    int garageCnt = expresswayCnt * 2;
    garages = (garage*)malloc(sizeof(garage) * garageCnt); // Allocate memory for global
    bestPerm = (int*)malloc(sizeof(int) * garageCnt); // Allocate memory for global

    // Input garages
    for (int i = 0; i < garageCnt; i++) {
        scanf("%d %d %s", &garages[i].x, &garages[i].y, garages[i].name);
    }

    int perm[garageCnt]; // Local array for current permutation
    int used[garageCnt]; // Local array to track used garages
    float bestValue = 9999999; // Initialize best value with a large number

    // Initialize used array
    for (int i = 0; i < garageCnt; i++) {
        used[i] = 0;
    }

    // Call the recursive permutation function
    perms(perm, used, 0, garageCnt, &bestValue, 0.0);

    // Output the best solution
    printf("%.3f\n", bestValue);
    for (int i = 0; i < garageCnt; i += 2) {
        printf("(%.3f, %.3f) -> (%.3f, %.3f)\n", garages[bestPerm[i]].x, garages[bestPerm[i]].y, garages[bestPerm[i+1]].x, garages[bestPerm[i+1]].y);
    }

    // Free allocated memory
    free(garages);
    free(bestPerm);

    return 0;
}

```

Explanation of Changes:

1. Global Variables:

- `garage* garages` : Holds the array of garage structures globally.
- `int* bestPerm` : Holds the best permutation globally.

2. Local Variables:

- `perm[]`, `used[]`, and all other variables are now locally scoped within functions.

3. Recursion and Early Pruning:

Maintained the recursive structure with the `used` array and early pruning if the partial sum exceeds the best distance found so far.

4. Memory Management:

We dynamically allocate memory for the two global variables (`garages` and `bestPerm`) and free them at the end to avoid memory leaks.

- **HOW CHATGPT HELPED ME**

Other than the fact that ChatGPT has helped me reorganize my code and moved certain functions and calculations to minimize the number of pointers and parameters, ChatGPT mostly helped me with the way the perm function handles its calculations. Something that ChatGPT optimized was pruning branches early when the current distance sum got larger than the best distance sum. This significantly lowers the amount of permutations that are evaluated so that the code can run quicker and more efficiently. Another thing that ChatGPT helped me implement was the use of global variables. This made it possible to lower the pointers and parameters that I had in my previous code.