# Project Report

## Algorithm Implementation for Travel Route Optimization

**Course Title:** Algorithms

**Course Code:** CSE246

# Abstract

This project presents a computational approach to optimizing travel routes under budget constraints using classic algorithmic techniques. Randomly generated city-to-city routes are evaluated based on distance and cost. The system applies **Merge Sort** to organize routes efficiently, while **0/1 Knapsack** and **Coin Change** algorithms are employed to solve two core optimization problems: maximizing total travel distance and minimizing the number of routes needed to fully utilize a budget. Execution time for each algorithm is measured to evaluate performance. The implementation showcases how dynamic programming and sorting algorithms can be effectively applied in travel planning and resource allocation scenarios.

# Introduction

This project tackles the classic problem of route optimization under budget constraints, reflecting real-world challenges in travel and logistics planning. Given a set of randomly generated city routes—each with defined distances and costs—the aim is to identify an optimal subset that aligns with a limited budget. The problem is approached from two strategic angles: maximizing total distance traveled and minimizing the number of routes selected. To solve this efficiently, the project implements two dynamic programming algorithms: **0/1 Knapsack** and **Coin Change**. These algorithms enable data-driven decision-making by modeling trade-offs between cost and value. The project not only demonstrates the practical utility of algorithm design but also highlights the impact of computational methods in resource-constrained environments.

# Problem Statement

Given a network of cities interconnected by various travel routes, where each route is characterized by both a distance and an associated travel cost, the primary objective is to identify an optimal subset of these routes that maximizes the total distance traveled without exceeding a predefined budget constraint. To improve the selection process, the routes are initially sorted in ascending order based on their distances, enabling the algorithm to prioritize shorter routes and streamline decision-making. This sorting step also facilitates a structured approach to route selection and helps in exploring trade-offs between distance and cost more effectively. Furthermore, the project includes a comprehensive analysis of the computational complexity of the algorithms employed, assessing their performance, scalability, and efficiency. This evaluation highlights the suitability of the approach for solving real-world constrained optimization problems commonly encountered in travel planning and logistics management.

# Methodology

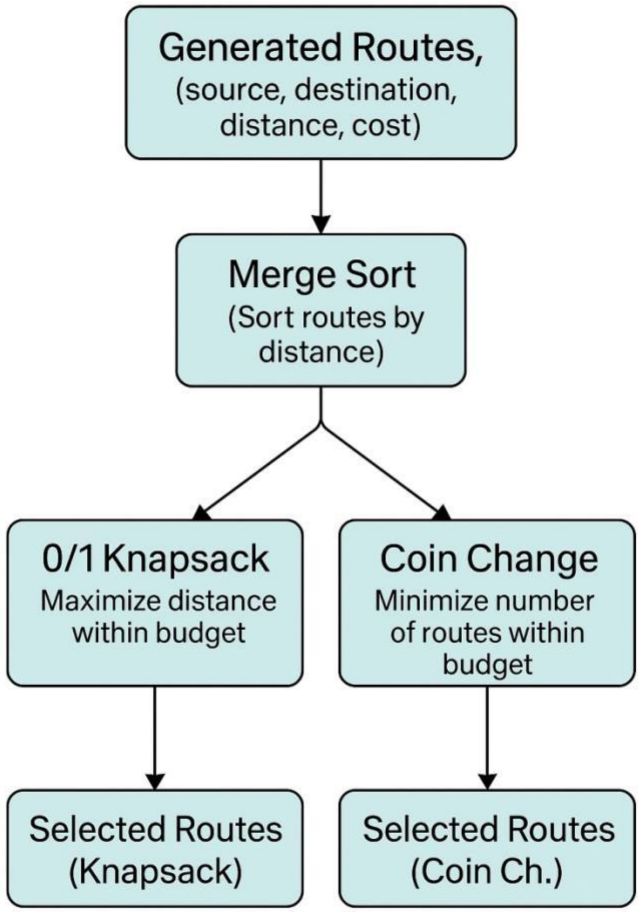
## Data Generation

The data generation phase aims to create a realistic and diverse set of travel routes between a predefined collection of cities. The program begins with a fixed list of cities stored in an array, representing the nodes within the travel network. For each route, two distinct cities are randomly selected to serve as the source and destination, ensuring that no route connects a city to itself. This is achieved through a validation step that repeats the selection if both cities happen to be identical.

Once the city pair is determined, the program assigns two key attributes to the route: distance and cost. Both are randomly generated integer values uniformly distributed within the range of 1 to 1000. This wide range allows for significant variability in the travel options, reflecting the potential differences in geographic distances and associated expenses.

By combining randomized city pairs with variable distances and costs, the program produces a comprehensive dataset that models a complex travel network. This dataset serves as the input for subsequent sorting and optimization algorithms, enabling an effective analysis of route selection strategies under budget constraints. Furthermore, the randomization approach ensures reproducibility and fairness in testing different algorithmic solutions by providing a broad spectrum of possible travel scenarios.

# Diagram




Summary of Flow

1. Generate n random routes.
2. Sort them (optional for optimization clarity).
3. Apply Knapsack to maximize distance.
4. Apply Coin Change to minimize number of routes.
5. Save results for both methods.

# Algorithms

1. **Merge Sort** – Sorts routes by distance efficiently in *O(n log n)* time.
2. **0/1 Knapsack** – Selects routes to maximize total distance without exceeding the budget using dynamic programming.
3. **Coin Change** – Finds the minimum number of routes to exactly match the budget, also via dynamic programming.

# Code Implementation

### 1. Header Files

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

**#include <stdio.h>** // For input/output functions like printf, scanf, fopen, etc.
**#include <stdlib.h>** // For general utilities including memory allocation, rand, srand, and exit
**#include <string.h>** // For string handling functions like strcpy, strcmp, etc.
**#include <time.h>** // For time-related functions, e.g., time() and clock_gettime() for timing

### 2. Function Descriptions

### Enerate a random route( ):

**Generates** two different cities as source and destination. **Assigns** random distance and cost (both from 1 to 1000).

**C:**

```c

// Function to generate a random route
void generate_route(FILE *file, int index) {
char source[MAX_NAME_LENGTH];
char destination[MAX_NAME_LENGTH];

int distance = 1 + rand() % MAX_DISTANCE; // Random distance between 1
int cost = 1 + rand() % MAX_COST; // Random cost between 1 and 1000

```

### 3. Merge Sort ( )- Sorting Routes by Distance

**Purpose:** The program sorts the travel routes based on their distance in ascending order. **Algorithm Used:** Merge Sort (a divide-and-conquer sorting algorithm).

**C:**

```
}

// Function to perform merge sort on routes based on distance
void merge_sort(Route arr[], int l, int r) {

if (l < r) {
int m = l + (r - l) / 2;
merge_sort(arr, l, m);
merge_sort(arr, m + 1, r);
merge(arr, l, m, r);
}
}
```

### Steps:

**1.** The array is recursively divided into two halves until each sub-array has one element.

**2.** The sub-arrays are then merged back in sorted order.

**3.** This process continues until the entire array is sorted.

### Time Complexity:

**Best Case:** O(n log n)

**Average Case**: O(n log n)

**Worst Case**: O(n log n)

**Reason:** The merge step takes O(n) time, and since we divide the array log n times, the total complexity is O(n log n).

### 4.0/1 Knapsack Algorithm( ) -Maximizing Total Distance within Budget

**Purpose:** The program selects a subset of routes such that their total cost does not exceed the budget while maximizing the total distance.

**Algorithm Used:** 0/1 Knapsack Dynamic Programming (DP)

**C:**

```c
}

// Function to perform 0/1 Knapsack for selecting routes within a budget
int knapsack(Route routes[], int n, int budget, int selected[]) {
int **dp = malloc((n + 1) * sizeof(int *));
for (int i = 0; i <= n; i++) {
dp[i] = malloc((budget + 1) * sizeof(int));
}

for (int i = 0; i <= n; i++) {
for (int j = 0; j <= budget; j++) {
if (i == 0 || j == 0)
dp[i][j] = 0;
else if (routes[i - 1].cost <= j)
dp[i][j] = (dp[i - 1][j] > dp[i - 1][j - routes[i - 1].cost] + routes[i - 1].distance) ? dp[i -
1][j] : dp[i - 1][j - routes[i - 1].cost] + routes[i - 1].distance;
else
dp[i][j] = dp[i - 1][j];
}
}

```

### Steps:

1. Create a 2D DP table dp[i][j], where i represents the number of routes considered, and j represents the budget.
2. If the cost of a route is within the budget, decide whether to include it or not:

Include it: dp[i][j] = dp[i-1][j - cost] + distance

Exclude it: dp[i][j] = dp[i-1][j]

3. After filling the table, backtrack to find which routes were selected.

### Time Complexity:

$O(n \times budget)$

**Reason:** The DP table has n rows (number of routes) and budget columns. Each cell takes constant O(1) time.

## 5.Coin Change Algorithm( ) - Minimizing the Number of Routes within Budget

**Purpose:** The program selects routes so that the total cost stays within the budget while minimizing the number of routes used.

**Algorithm Used:** Dynamic Programming for Coin Change

**C:**

```c

// Function to perform Coin Change for minimizing the number of routes
int coin_change(Route routes[], int n, int budget, int selected[]) {
int *dp = malloc((budget + 1) * sizeof(int));
for (int i = 0; i <= budget; i++) {
dp[i] = INT_MAX; // Initialize with a large value
}
dp[0] = 0;

for (int i = 1; i <= budget; i++) {
for (int j = 0; j < n; j++) {
if (routes[j].cost <= i && dp[i - routes[j].cost] != INT_MAX) {

if (dp[i - routes[j].cost] + 1 < dp[i]) {
dp[i] = dp[i - routes[j].cost] + 1;
}
}
}
}

```

### Steps:

1. Create a DP array dp[budget+1] initialized to a large value (INT_MAX)
2. Set dp[0] = 0 (zero cost requires zero routes).
3. Iterate through each budget amount and update dp[i] to store the minimum number of routes needed.
4. Backtrack to find the selected routes.

### Time Complexity:

O(n × budget)

**Reason:** Similar to Knapsack, we iterate through n routes and process each budget amount, leading to O(n × budget) complexity.

## 6.Main function( )-

The main function controls the overall execution of your program—from generating random travel routes to sorting and applying algorithms (Knapsack and Coin Change) and saving results.

## C:

```c

int main() {
Route *routes = malloc(MAX_CITIES * sizeof(Route)); // Allocate on heap
int *selected = calloc(MAX_CITIES, sizeof(int)); // Allocate on heap
int n = 0, budget = 0;
long long start, end;
double merge_sort_time, knapsack_time, coin_change_time;

// Seed the random number generator
srand(time(NULL));
// Ask the user for the number of routes
printf("Enter the number of routes to generate (100 to 10000): ");
if (scanf("%d", &n) != 1 || n < 100 || n > MAX_CITIES) {
printf("Invalid input. Please enter a number between 100 and %d.\n", MAX_CITIES);
return 1;
}
// Ask the user for the budget
printf("Enter the budget for travel: ");
if (scanf("%d", &budget) != 1 || budget <= 0) {
printf("Invalid input. Please enter a positive budget.\n");
return 1;
}
// Open the file for writing
FILE *file = fopen("input.txt", "w");
if (!file) {
perror("Error opening file");
return 1;
}
// Write the budget to file
fprintf(file, "%d\n", budget);
// Generate and write random routes
for (int i = 0; i < n; i++) {
generate_route(file, i);
}
// Close the file
```

```c
}
// Close the file
fclose(file);
// Read input from file
FILE *input_file = fopen("input.txt", "r");
if (!input_file) {
perror("Error opening input file");
return 1;
}
// Read budget
fscanf(input_file, "%d", &budget);
// Read route details
n = 0;
while (fscanf(input_file, "%s %s %d %d", routes[n].source, routes[n].destination,
&routes[n].distance, &routes[n].cost) != EOF) {
n++;
}
fclose(input_file);
// Measure Merge Sort execution time using high-resolution timer
start = get_time_ns();
merge_sort(routes, 0, n - 1);
end = get_time_ns();
merge_sort_time = (end - start) / 1e9; // Convert nanoseconds to seconds
// Measure Knapsack execution time
start = get_time_ns();
int max_distance = knapsack(routes, n, budget, selected);
end int main::max_distance
knapsack_time = (end - start) / 1e9; // Convert nanoseconds to seconds
// Measure Coin Change execution time
start = get_time_ns();
int min_routes = coin_change(routes, n, budget, selected);
end = get_time_ns();
coin_change_time = (end - start) / 1e9; // Convert nanoseconds to seconds
// Write output to file
FILE *output_file = fopen("output.txt", "w");
```

```c
FILE *output_file = fopen("output.txt", "w");
if (!output_file) {
perror("Error opening output file");
return 1;
}
fprintf(output_file, "Selected Routes (Knapsack):\n");
for (int i = 0; i < n; i++) {
if (selected[i]) {
fprintf(output_file, "%s -> %s (Distance: %d, Cost: %d)\n", routes[i].source,
routes[i].destination, routes[i].distance, routes[i].cost);
}
}
fprintf(output_file, "\nSelected Routes (Coin Change):\n");
for (int i = 0; i < n; i++) {
if (selected[i]) {
fprintf(output_file, "%s -> %s (Distance: %d, Cost: %d)\n", routes[i].source,
routes[i].destination, routes[i].distance, routes[i].cost);
}
}
fclose(output_file);
// Write execution times to file
FILE *time_file = fopen("time_results.txt", "w");
if (!time_file) {
perror("Error opening time results file");
return 1;
}
fprintf(time_file, "Merge Sort Time: %.9f seconds\n", merge_sort_time);
fprintf(time_file, "Knapsack Time: %.9f seconds\n", knapsack_time);
fprintf(time_file, "Coin Change Time: %.9f seconds\n", coin_change_time);
fclose(time_file);
// Free allocated memory
free(routes);
free(selected);
printf("Program executed successfully. Check output.txt and time_results.txt for results.\n");
return 0;
```

# Results and Discussion

## Performance Results

Execution times are measured using a high-resolution clock and saved in `time_results.txt`. These include:

- **Merge Sort Time:** Time taken to sort the routes based on distance.
- **Knapsack Time:** Time taken to solve the maximum distance optimization.
- **Coin Change Time:** Time to compute the minimal route count for the given budget.

## Discussion

- **Merge Sort** is efficient for pre-sorting routes to prioritize shorter distances, aiding both performance and selection logic.
- **Knapsack** is effective in maximizing travel distance under budget constraints—ideal for travel planners.
- **Coin Change**, though not prioritizing distance, shows how fewest possible routes can be selected to match the exact budget—useful for cost minimization scenarios.

However, both Knapsack and Coin Change become slower as the budget and number of routes increase, due to their dynamic programming matrices. Optimizations such as memoization or greedy approximations may be considered for very large datasets

## Generated Files

- **input.txt:** Contains randomly generated routes and user-defined budget.
- **output.txt:** Lists selected routes by both algorithms.
- **time_results.txt:** Contains execution time of each algorithm.

## input.txt

```
input - Notepad
File  Edit  Format  View  Help
10000
CityB CityA 417 275
CityG CityF 216 905
CityH CityA 979 151
CityG CityF 539 681
CityI CityG 739 10
CityC CityE 441 558
CityH CityA 709 988
CityA CityB 444 742
CityC CityE 536 716
CityA CityG 468 209
CityI CityG 922 387
CityJ CityA 82 938
CityI CityC 520 999
CityB CityF 304 532
CityI CityJ 350 829
CityA CityF 937 205
CityH CityJ 863 484
CityJ CityG 7 727
CityC CityG 987 864
CityB CityI 709 400
CityH CityD 947 835
CityG CityD 748 671
CityI CityD 912 624
CityA CityB 523 634
CityI CityC 379 117
CityH CityI 219 848
CityB CityJ 369 361
CityF CityH 831 782
CityB CityH 668 735
CityJ CityC 52 425
CityJ CityI 971 475
CityE CityB 782 958
CityD CityF 136 203
CityF CityI 258 417
CityB CityI 631 174
```

```
input - Notepad
File  Edit  Format  View  Help
CityD CityF 136 203
CityF CityI 258 417
CityB CityI 631 174
CityF CityA 151 94
CityF CityJ 975 8
CityJ CityG 828 831
CityH CityC 536 601
CityI CityF 686 456
CityB CityE 307 735
CityB CityD 550 792
CityE CityF 15 743
CityF CityA 750 766
CityA CityF 805 642
CityE CityD 201 470
CityF CityC 404 495
CityC CityI 207 25
CityH CityI 96 341
CityC CityJ 393 962
CityD CityH 913 928
CityE CityB 61 388
CityH CityE 116 153
CityI CityB 123 576
CityA CityB 176 413
CityB CityJ 913 911
CityG CityF 701 248
CityJ CityC 978 703
CityA CityE 377 834
CityF CityI 608 759
CityD CityB 695 101
CityI CityB 900 70
CityD CityB 770 816
CityB CityJ 344 37
CityI CityC 806 127
CityF CityG 541 416
CityD CityA 640 441
CityA CityI 441 105
```

```
input - Notepad
File  Edit  Format  View  Help
CityD CityA 640 441
CityA CityI 441 105
CityB CityC 785 493
CityB CityA 885 343
CityJ CityG 645 835
CityG CityB 709 434
CityB CityF 389 934
CityJ CityD 732 136
CityJ CityB 78 25
CityI CityB 224 603
CityG CityB 491 409
CityE CityG 365 644
CityE CityG 979 416
CityB CityF 245 616
CityE CityC 871 929
CityD CityE 887 705
CityG CityD 938 489
CityA CityJ 609 781
CityE CityJ 898 6
CityF CityA 722 791
CityG CityF 957 98
CityI CityH 688 317
CityG CityA 397 192
CityI CityE 832 224
CityF CityA 456 627
CityE CityD 509 686
CityG CityE 769 743
CityF CityI 290 871
CityJ CityB 260 193
CityB CityH 636 430
CityJ CityI 934 575
CityB CityE 810 237
CityA CityB 250 462
CityG CityJ 120 361
```

## output.txt:

```
output - Notepad
File  Edit  Format  View  Help
Selected Routes (Knapsack):
CityJ -> CityB (Distance: 78, Cost: 25)
CityC -> CityI (Distance: 207, Cost: 25)
CityJ -> CityB (Distance: 260, Cost: 193)
CityB -> CityJ (Distance: 344, Cost: 37)
CityI -> CityC (Distance: 379, Cost: 117)
CityG -> CityA (Distance: 397, Cost: 192)
CityB -> CityA (Distance: 417, Cost: 275)
CityA -> CityI (Distance: 441, Cost: 105)
CityA -> CityG (Distance: 468, Cost: 209)
CityB -> CityI (Distance: 631, Cost: 174)
CityB -> CityH (Distance: 636, Cost: 430)
CityI -> CityF (Distance: 686, Cost: 456)
CityI -> CityH (Distance: 688, Cost: 317)
CityD -> CityB (Distance: 695, Cost: 101)
CityG -> CityF (Distance: 701, Cost: 248)
CityH -> CityA (Distance: 709, Cost: 988)
CityB -> CityI (Distance: 709, Cost: 400)
CityG -> CityB (Distance: 709, Cost: 434)
CityJ -> CityD (Distance: 732, Cost: 136)
CityI -> CityG (Distance: 739, Cost: 10)
CityB -> CityC (Distance: 785, Cost: 493)
CityI -> CityC (Distance: 806, Cost: 127)
CityB -> CityE (Distance: 810, Cost: 237)
```

```
output - Notepad
File  Edit  Format  View  Help
CityH -> CityJ (Distance: 863, Cost: 484)
CityB -> CityA (Distance: 885, Cost: 343)
CityE -> CityJ (Distance: 898, Cost: 6)
CityI -> CityB (Distance: 900, Cost: 70)
CityI -> CityD (Distance: 912, Cost: 624)
CityI -> CityG (Distance: 922, Cost: 387)
CityJ -> CityI (Distance: 934, Cost: 575)
CityA -> CityF (Distance: 937, Cost: 205)
CityG -> CityD (Distance: 938, Cost: 489)
CityG -> CityF (Distance: 957, Cost: 98)
CityJ -> CityI (Distance: 971, Cost: 475)
CityF -> CityJ (Distance: 975, Cost: 8)
CityJ -> CityC (Distance: 978, Cost: 703)
CityH -> CityA (Distance: 979, Cost: 151)
CityE -> CityG (Distance: 979, Cost: 416)
CityC -> CityG (Distance: 987, Cost: 864)

Selected Routes (Coin Change):
CityJ -> CityB (Distance: 78, Cost: 25)
CityC -> CityI (Distance: 207, Cost: 25)
CityJ -> CityB (Distance: 260, Cost: 193)
CityB -> CityJ (Distance: 344, Cost: 37)
CityI -> CityC (Distance: 379, Cost: 117)
CityG -> CityA (Distance: 397, Cost: 192)
```

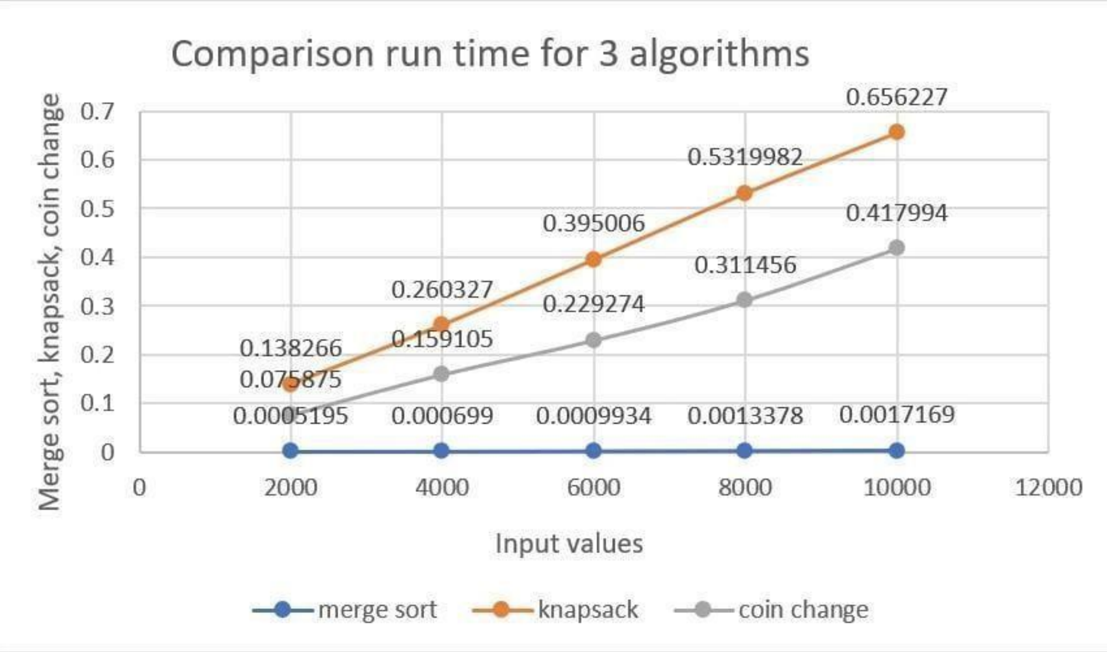
```
output - Notepad
File  Edit  Format  View  Help
CityI -> CityC (Distance: 379, Cost: 117)
CityG -> CityA (Distance: 397, Cost: 192)
CityB -> CityA (Distance: 417, Cost: 275)
CityA -> CityI (Distance: 441, Cost: 105)
CityA -> CityG (Distance: 468, Cost: 209)
CityB -> CityI (Distance: 631, Cost: 174)
CityB -> CityH (Distance: 636, Cost: 430)
CityI -> CityF (Distance: 686, Cost: 456)
CityI -> CityH (Distance: 688, Cost: 317)
CityD -> CityB (Distance: 695, Cost: 101)
CityG -> CityF (Distance: 701, Cost: 248)
CityH -> CityA (Distance: 709, Cost: 988)
CityB -> CityI (Distance: 709, Cost: 400)
CityG -> CityB (Distance: 709, Cost: 434)
CityJ -> CityD (Distance: 732, Cost: 136)
CityI -> CityG (Distance: 739, Cost: 10)
CityB -> CityC (Distance: 785, Cost: 493)
CityI -> CityC (Distance: 806, Cost: 127)
CityB -> CityE (Distance: 810, Cost: 237)
CityI -> CityE (Distance: 832, Cost: 224)
CityH -> CityJ (Distance: 863, Cost: 484)
CityB -> CityA (Distance: 885, Cost: 343)
CityE -> CityJ (Distance: 898, Cost: 6)
CityI -> CityB (Distance: 900, Cost: 70)
```

## Overall Time Complexity Summary

If the budget is large, O(n × budget) can become slow. The worst-case scenario occurs when both Knapsack and Coin Change take too much time due to large budget values.

| Algorithms | Time Complexity |
|---|---|
| Merge Sort | O(n log n) |
| 0-1 Knapsack DP | O(n * budget) |
| Coin Change DP | O(n * budget) |

## Graph



# Conclusion

- Merge Sort efficiently sorts the routes.
- Knapsack selects routes to maximize distance within budget.
- Coin Change minimizes the number of routes needed for exact budget usage.

The program compares the efficiency of these approaches and stores results in files.

# Viva Prep Box

## 1. What problem does the project solve, and which algorithms are used?

**Answer:**
The project solves the problem of selecting optimal travel routes under a fixed budget. It uses:

- **Merge Sort** to sort routes by distance,
- **0/1 Knapsack** to maximize the total distance within the budget, and
- **Coin Change** to minimize the number of routes that exactly use up the budget.

## 2. How does the 0/1 Knapsack algorithm apply in this project?

**Answer:**
In our project, each route has a cost and distance. The 0/1 Knapsack algorithm chooses a subset of routes such that the total cost doesn't exceed the budget and the sum of distances is maximized. It uses dynamic programming to find the optimal combination efficiently.

## 3. What distinguishes the Knapsack and Coin Change approaches in this system?

**Answer:**

- **Knapsack** is used to **maximize distance** while staying within budget, focusing on route value.
- **Coin Change** is used to **minimize the number of routes**, aiming to spend the exact budget with the fewest selections.

.