**Program 9:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to perform selection sort on an array

void selectionSort(int arr[], int n)

{

int i, j, min\_idx;

for (i = 0; i < n-1; i++)

{

min\_idx = i; // Assume the current element is the minimum

for (j = i+1; j < n; j++)

{

if (arr[j] < arr[min\_idx])

{

min\_idx = j; // Update min\_idx if a smaller element is found

}

}

// Swap the found minimum element with the current element

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

// Function to generate an array of random numbers

void generateRandomNumbers(int arr[], int n)

{

for (int i = 0; i < n; i++)

{

arr[i] = rand() % 10000; // Generate random numbers between 0 and 9999

}

}

int main()

{

int n;

printf("Enter number of elements: ");

scanf("%d", &n); // Read the number of elements from the user

if (n <= 5000)

{

printf("Please enter a value greater than 5000\n");

return 1; // Exit if the number of elements is not greater than 5000

}

// Allocate memory for the array

int \*arr = (int \*)malloc(n \* sizeof(int));

if (arr == NULL)

{

printf("Memory allocation failed\n");

return 1; // Exit if memory allocation fails

}

// Generate random numbers and store them in the array

generateRandomNumbers(arr, n);

// Measure the time taken to sort the array

clock\_t start = clock();

selectionSort(arr, n);

clock\_t end = clock();

// Calculate and print the time taken to sort the array

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken to sort %d elements: %f seconds\n", n, time\_taken);

// Free the allocated memory

free(arr);

return 0;

}

**Run the Program for Various Values of n**

To collect data, run the program with different values of n greater than 5000, such as 6000, 7000, 8000, etc., and record the time taken for each.

Output:

Enter number of elements: 6000

Time taken to sort 6000 elements: 0.031000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 7000

Time taken to sort 7000 elements: 0.034000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 8000

Time taken to sort 8000 elements: 0.047000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 9000

Time taken to sort 9000 elements: 0.052000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 10000

Time taken to sort 10000 elements: 0.077000 seconds

**Plot the Results**

You can use a graphing tool like [Python](https://vtucode.in/bcsl404-program-9/) with matplotlib to plot the results.

*import* matplotlib.pyplot *as* plt

*# data collected*

n\_values = [6000, 7000, 8000, 9000, 10000]

time\_taken = [0.031000, 0.034000, 0.047000, 0.052000, 0.077000] *# replace with actual times recorded*

plt.plot(n\_values, time\_taken, *marker*='o')

plt.title('Selection Sort Time Complexity')

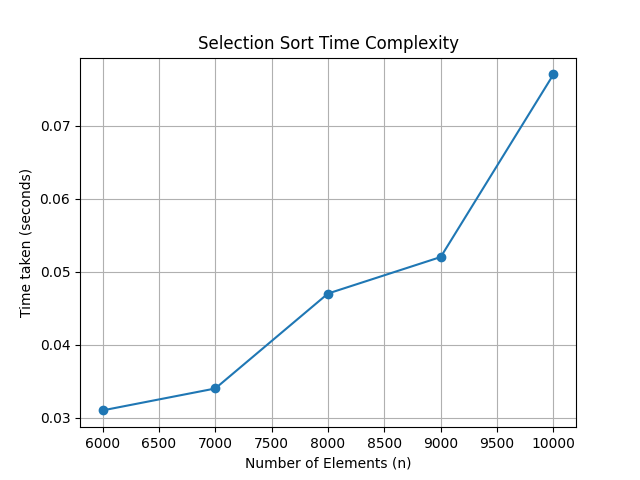
plt.xlabel('Number of Elements (n)')

plt.ylabel('Time taken (seconds)')

plt.grid(True)

plt.show()

output:



Pgm 10:

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <time.h>

*// Function to swap two elements*

void swap(int\* *a*, int\* *b*)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

*// Partition function for Quick Sort*

int partition(int *arr*[], int *low*, int *high*)

{

int pivot = arr[high]; *// Pivot element*

int i = (low - 1); *// Index of smaller element*

*for* (int j = low; j <= high - 1; j++)

{

*if* (arr[j] < pivot)

{

i++; *// Increment index of smaller element*

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

*return* (i + 1);

}

*// Quick Sort function*

void quickSort(int *arr*[], int *low*, int *high*)

{

*if* (low < high)

{

int pi = partition(arr, low, high);

*// Recursively sort elements before and after partition*

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

*// Function to generate random numbers*

void generateRandomNumbers(int *arr*[], int *n*)

{

*for* (int i = 0; i < n; i++)

{

arr[i] = rand() % 100000; *// Generate random numbers between 0 and 99999*

}

}

int main()

{

int n;

printf("Enter number of elements: ");

scanf("%d", &n); *// Read the number of elements from the user*

*if* (n <= 5000)

{

printf("Please enter a value greater than 5000\n");

*return* 1; *// Exit if the number of elements is not greater than 5000*

}

*// Allocate memory for the array*

int \*arr = (int \*)malloc(n \* sizeof(int));

*if* (arr == NULL)

{

printf("Memory allocation failed\n");

*return* 1; *// Exit if memory allocation fails*

}

*// Generate random numbers and store them in the array*

generateRandomNumbers(arr, n);

*// Measure the time taken to sort the array*

clock\_t start = clock();

quickSort(arr, 0, n - 1);

clock\_t end = clock();

*// Calculate and print the time taken to sort the array*

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken to sort %d elements: %f seconds\n", n, time\_taken);

*// Free the allocated memory*

free(arr);

*return* 0;

}

Output:

Enter number of elements: 10000

Time taken to sort 10000 elements: 0.0000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 20000

Time taken to sort 20000 elements: 0.015000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 30000

Time taken to sort 30000 elements: 0.011000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 35000

Time taken to sort 35000 elements: 0.003000 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 50000

Time taken to sort 50000 elements: 0.015000 seconds

**Plot the Results**

You can use a graphing tool like [Python](https://vtucode.in/bcsl404-program-9/) with matplotlib to plot the results.

*import* matplotlib.pyplot *as* plt

*# Example data collected*

n\_values = [10000, 20000, 30000, 35000, 50000]

time\_taken = [0.0000, 0.015000, 0.011000, 0.003000, 0.015000] *# replace with actual times recorded*

plt.plot(n\_values, time\_taken, *marker*='o')

plt.title('Quick Sort Time Complexity')

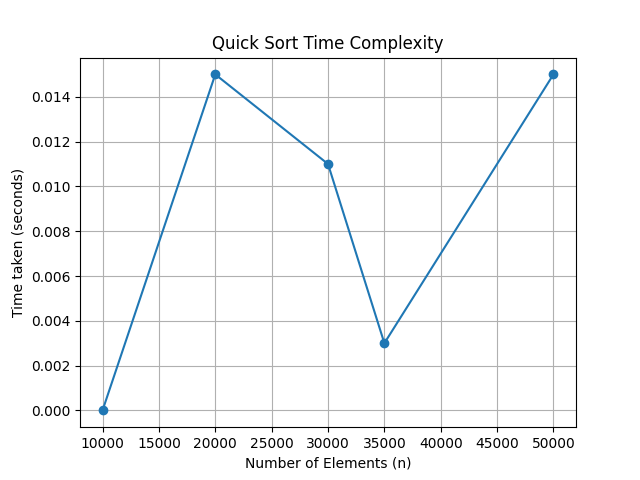
plt.xlabel('Number of Elements (n)')

plt.ylabel('Time taken (seconds)')

plt.grid(True)

plt.show()

output:



Pgm 11:

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <time.h>

*// Function to merge two sorted arrays*

void merge(int *arr*[], int *left*, int *mid*, int *right*)

{

int i, j, k;

int n1 = mid - left + 1;

int n2 = right - mid;

int \*L = (int \*)malloc(n1 \* sizeof(int));

int \*R = (int \*)malloc(n2 \* sizeof(int));

*for* (i = 0; i < n1; i++)

L[i] = arr[left + i];

*for* (j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

i = 0;

j = 0;

k = left;

*while* (i < n1 && j < n2)

{

*if* (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

*else*

{

arr[k] = R[j];

j++;

}

k++;

}

*while* (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

*while* (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

free(L);

free(R);

}

*// Function to implement Merge Sort*

void mergeSort(int *arr*[], int *left*, int *right*)

{

*if* (left < right)

{

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

*// Function to generate random integers*

void generateRandomArray(int *arr*[], int *n*)

{

*for* (int i = 0; i < n; i++)

arr[i] = rand() % 100000; *// Generate random integers between 0 and 99999*

}

int main()

{

int n;

printf("Enter the number of elements: ");

scanf("%d", &n);

*if* (n <= 5000)

{

printf("Please enter a value greater than 5000\n");

*return* 1; *// Exit if the number of elements is not greater than 5000*

}

int \*arr = (int \*)malloc(n \* sizeof(int));

*if* (arr == NULL)

{

printf("Memory allocation failed\n");

*return* 1; *// Exit if memory allocation fails*

}

generateRandomArray(arr, n);

*// Repeat the sorting process multiple times to increase duration for timing*

clock\_t start = clock();

*for* (int i = 0; i < 1000; i++)

{

mergeSort(arr, 0, n - 1);

}

clock\_t end = clock();

*// Calculate the time taken for one iteration*

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC / 1000.0;

printf("Time taken to sort %d elements: %f seconds\n", n, time\_taken);

free(arr);

*return* 0;

}

**Run the Program for Various Values of n**

To collect data, run the program with different values of n greater than 5000, such as 6000, 7000, 8000, etc., and record the time taken for each.

Output:

Enter number of elements: 6000

Time taken to sort 6000 elements: 0.000709 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 7000

Time taken to sort 7000 elements: 0.000752 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 8000

Time taken to sort 8000 elements: 0.000916 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 9000

Time taken to sort 9000 elements: 0.001493 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 10000

Time taken to sort 10000 elements: 0.001589 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 11000

Time taken to sort 11000 elements: 0.002562 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 12000

Time taken to sort 12000 elements: 0.001944 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 13000

Time taken to sort 13000 elements: 0.002961 seconds

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of elements: 15000

Time taken to sort 15000 elements: 0.003563 seconds

*import* matplotlib.pyplot *as* plt

*# data collected (replace with actual data)*

n\_values = [6000, 7000, 8000, 9000, 10000, 11000, 12000, 13000, 15000]

time\_taken = [0.000709, 0.000752, 0.000916, 0.001493, 0.001589, 0.002562, 0.001944, 0.002961, 0.003563] *# Replace with actual times recorded*

plt.plot(n\_values, time\_taken, *marker*='o')

plt.title('Merge Sort Time Complexity')

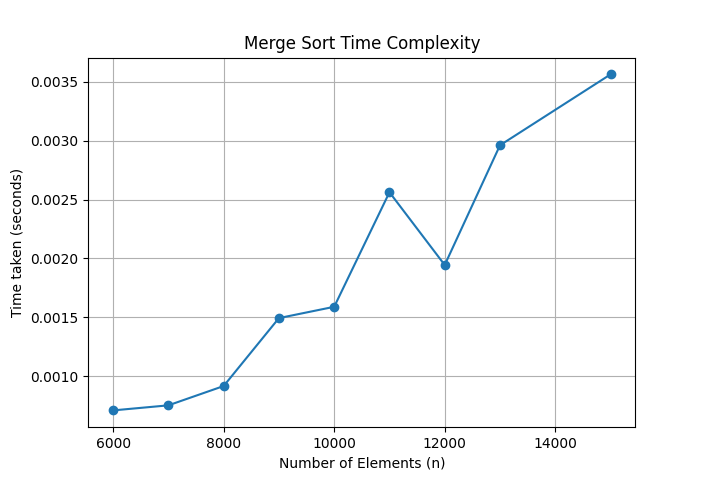
plt.xlabel('Number of Elements (n)')

plt.ylabel('Time taken (seconds)')

plt.grid(True)

plt.show()

output:



Pgm 12:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Function to print the solution

void printSolution(int \*\*board, int N)

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

printf("%s ", board[i][j] ? "Q" : "#");

}

printf("\n");

}

}

// Function to check if a queen can be placed on board[row][col]

bool isSafe(int \*\*board, int N, int row, int col)

{

int i, j;

// Check this row on left side

for (i = 0; i < col; i++)

{

if (board[row][i])

{

return false;

}

}

// Check upper diagonal on left side

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

{

if (board[i][j])

{

return false;

}

}

// Check lower diagonal on left side

for (i = row, j = col; j >= 0 && i < N; i++, j--)

{

if (board[i][j])

{

return false;

}

}

return true;

}

// A recursive utility function to solve N Queen problem

bool solveNQUtil(int \*\*board, int N, int col)

{

// If all queens are placed, then return true

if (col >= N)

{

return true;

}

// Consider this column and try placing this queen in all rows one by one

for (int i = 0; i < N; i++)

{

if (isSafe(board, N, i, col))

{

// Place this queen in board[i][col]

board[i][col] = 1;

// Recur to place rest of the queens

if (solveNQUtil(board, N, col + 1))

{

return true;

}

// If placing queen in board[i][col] doesn't lead to a solution,

// then remove queen from board[i][col]

board[i][col] = 0; // BACKTRACK

}

}

// If the queen cannot be placed in any row in this column col, then return false

return false;

}

// This function solves the N Queen problem using Backtracking

// It mainly uses solveNQUtil() to solve the problem

// It returns false if queens cannot be placed, otherwise, return true and prints the placement of queens

bool solveNQ(int N)

{

int \*\*board = (int \*\*)malloc(N \* sizeof(int \*));

for (int i = 0; i < N; i++)

{

board[i] = (int \*)malloc(N \* sizeof(int));

for (int j = 0; j < N; j++)

{

board[i][j] = 0;

}

}

if (!solveNQUtil(board, N, 0))

{

printf("Solution does not exist\n");

for (int i = 0; i < N; i++)

{

free(board[i]);

}

free(board);

return false;

}

printSolution(board, N);

for (int i = 0; i < N; i++)

{

free(board[i]);

}

free(board);

return true;

}

int main()

{

int N;

printf("Enter the number of queens: ");

scanf("%d", &N);

solveNQ(N);

return 0;

}

output:

Enter the number of queens: 4

# # Q #

Q # # #

# # # Q

# Q # #