Practical 2: Sorting Algorithms

Student: Dhruv Sachdeva, sachdevadhruv023@ce.du.ac.in

Introduction:

Explanation of automating technique:
The input size n will be provided through command-line arguments:

```
if (argc < 2) {
    printf("Usage: %s <n>\n", argv[0]);
    return 1;
}
int n = atoi(argv[1]);  // read n from command-line argument
```

Adding this boiler code in main of all codes will automate the process of getting time and plotting. After this we can make a python wrapper which will run the programs for use and plot them too

Merge Sort

Merge Sort is a divide-and-conquer algorithm. It recursively splits the array into halves until single elements remain, and then merges them in sorted order.

Algorithm 1 Merge Sort Algorithm

Require: Array $arr[0 \dots n-1]$, integer n
Ensure: Sorted array arr
  function MERGESORT($arr, left, right$)
    if $left < right$ **then**
      $mid \leftarrow (left + right)/2$
      MERGESORT($arr, left, mid$)
      MERGESORT($arr, mid + 1, right$)
      MERGE($arr, left, mid, right$)
    end if
  end function

- Divide the array into two halves.
- Recursively sort both halves.
- Merge the two sorted halves.

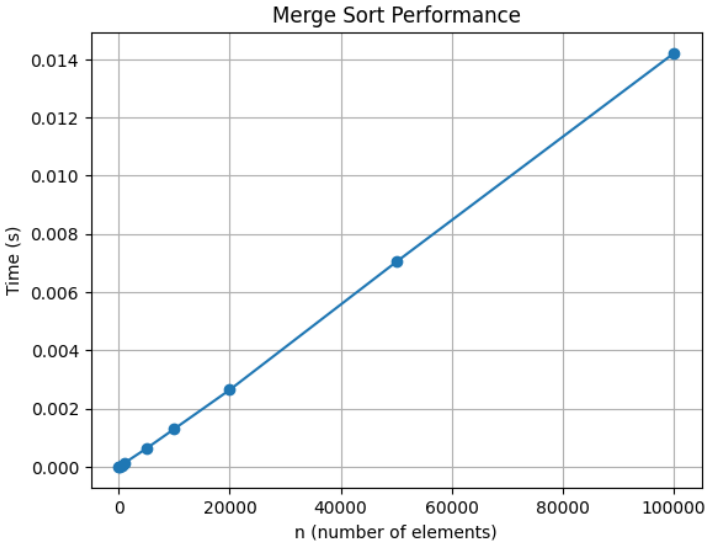
C Code:

```

#include <stdlib.h>
#include <stdio.h>
# include<time.h>// return clock ticks one clock tick in 1000
void merge(int arr[],int low, int mid, int high)
{
    int i,j,k;
```

```
8       int n1=mid-low+1;
9       int n2=high-mid;
10      int *L=(int*)malloc(n1*sizeof(int));
11      int *R=(int*)malloc(n2*sizeof(int));
12      for(i=0;i<n1;i++)
13      {
14          L[i]=arr[low+i];
15      }
16      for(j=0;j<n2;j++)
17      {
18          R[j]=arr[mid+1+j];
19      }
20      i=0;
21      j=0;
22      k=low;
23      while(i<n1 && j<n2)
24      {
25          if(L[i]<R[j])
26          {
27              arr[k]=L[i];
28              i++;
29          }
30          else
31          {
32              arr[k]=R[j];
33              j++;
34          }
35          k++;
36      }
37
38      while(i<n1)
39      {
40          arr[k]=L[i];
41          k++;i++;
42      }
43      while(j<n2)
44      {
45          arr[k]=R[j];
46          k++;j++;
47      }
48      free(L);
49      free(R);
50  }
51
52  void mergesort(int arr[],int low,int high)
53  {
54      if(low<high)
55      {
56          int mid=low+(high-low)/2; // to not exceed the integer limit of syst
57          mergesort(arr,low,mid);
58          mergesort(arr,mid+1,high);
```

```
59          merge(arr,low,mid,high);
60      }
61  }
62
63  void generateRandomArray(int arr[],int n)
64  {
65      for(int i=0;i<n;i++)
66      {
67          arr[i]=rand()%100000;//generate random integers
68      }
69  }
70  int main(int argc, char* argv[])
71  {
72      if (argc < 2) {
73          printf("Usage: %s <n>\n", argv[0]);
74          return 1;
75      }
76
77      int n = atoi(argv[1]);  // read n from command-line argument
78
79
80      int* arr=(int*)malloc(n*sizeof(int));
81      if(arr==NULL)
82      {
83          printf("Memory alloation failed!\n");
84          return 1;
85      }
86      generateRandomArray(arr,n);
87      clock_t start=clock();
88      for(int i=0;i<1000;i++)
89      {
90          mergesort(arr,0,n-1);
91      }
92      clock_t end=clock();
93      double time_taken=((double)(end-start))/CLOCKS_PER_SEC/1000.0;//time for
94      printf("%f", time_taken);
95      free(arr);
96
97      return 0;
98  }
```

Graph:



Algorithmic Analysis

- **Best Case:** Always divides and merges:

$$T_{best}(n) = O(n \log n)$$

- **Worst Case:** Same as best (recursively divides and merges):

$$T_{worst}(n) = O(n \log n)$$

- **Average Case:** On average, the merging dominates:

$$T_{avg}(n) = O(n \log n)$$

- **Space Complexity:** Extra space needed for merging:

$$S(n) = O(n)$$

Quick Sort

Quick Sort is a divide-and-conquer algorithm. It selects a pivot, partitions the array around it, and recursively sorts the subarrays.

Algorithm 2 Quick Sort Algorithm

Require: Array $arr[0 \ldots n-1]$, integer n
Ensure: Sorted array arr

function QUICKSORT($arr, low, high$)
  if $low < high$ **then**
    $p \leftarrow$ PARTITION($arr, low, high$)
    QUICKSORT($arr, low, p-1$)
    QUICKSORT($arr, p+1, high$)
  end if
end function

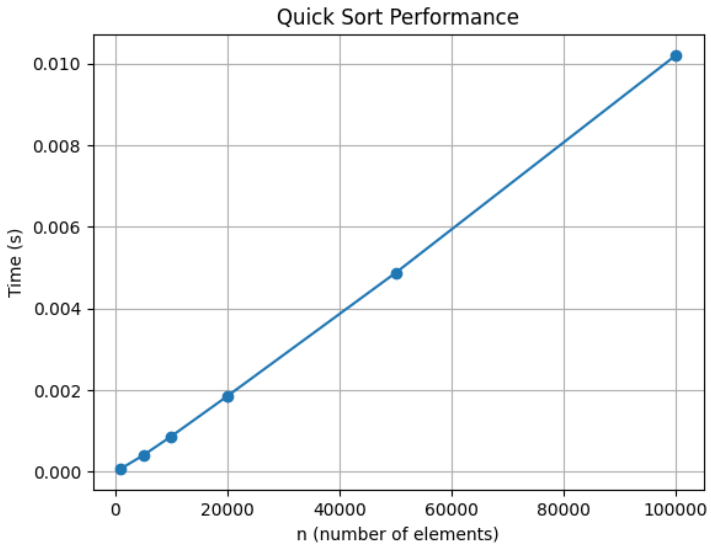
- Choose a pivot element.
- Partition the array around the pivot.
- Recursively apply quicksort to subarrays.

C Code:

```c

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

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[],int low,int high)
{
    int pivot=arr[high];
   int i=low-1;
   for (int j=low;j<high;j++)
   {
        if(arr[j]<pivot)
        {
            i++;
            swap(&arr[i],&arr[j]);
        }
    }
    swap(&arr[i+1],&arr[high]);
   return i+1;
}
void quicksort(int arr[],int low,int high)
```

```
{
    if (low<high)
    {
        int p=partition(arr,low,high);

    quicksort(arr,low,p-1);
    quicksort(arr,p+1,high);
}
}

void generateRandomArray(int arr[],int n)
{
    for(int i=0;i<n;i++)
    {
        arr[i]=rand()%100000;//generate random integers
    }
}


int main(int argc, char* argv[])
{
    if (argc < 2) {
        printf("Usage: %s <n>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);  // read n from command-line argument

    int* arr=(int*)malloc(n*sizeof(int));
    if(arr==NULL)
    {
        printf("Memory alloation failed!\n");
        return 1;
    }
    clock_t start=clock();
    for(int i=0;i<1000;i++)
    {
        generateRandomArray(arr,n);
        quicksort(arr,0,n-1);
    }
    clock_t end=clock();
    double time_taken=((double)(end-start))/CLOCKS_PER_SEC/1000.0;//time for
    printf("%f\n",time_taken);
    free(arr);

    return 0;
}
```

Graph:



Quick Sort using middle element

1. If the array has zero or one element, it is already sorted.
2. Select the middle element of the array as the pivot.
3. Partition the array into two subarrays:
   - Elements less than the pivot.
   - Elements greater than the pivot.
4. Recursively apply Quick Sort on the two subarrays.
5. Combine the results to obtain the sorted array.

Program Code (C)

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

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function using middle element as pivot
int partition(int arr[], int low, int high) {
    int mid = low + (high - low) / 2; // middle index
    int pivot = arr[mid];
    int i = low, j = high;

    while (i <= j) {
        while (arr[i] < pivot) i++;
```

```
        while (arr[j] > pivot) j--;
        if (i <= j) {
            swap(&arr[i], &arr[j]);
            i++;
            j--;
        }
    }
    return i;
}


void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi, high);
    }
}


int main() {
    int main(int argc, char* argv[])
{
    if (argc < 2) {
        printf("Usage: %s <n>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);  // read n from command-line argument

    int* arr=(int*)malloc(n*sizeof(int));
    if(arr==NULL)
    {
        printf("Memory alloation failed!\n");
        return 1;
    }
    clock_t start=clock();
    for(int i=0;i<1000;i++)
    {
        generateRandomArray(arr,n);
        quicksort(arr,0,n-1);
    }
    clock_t end=clock();
    double time_taken=((double)(end-start))/CLOCKS_PER_SEC/1000.0;//time for
    printf("%f\n",time_taken);
    free(arr);

    return 0;
}
```

Algorithmic Analysis

- **Best Case:** Pivot divides the array into equal halves:

$$T_{best}(n) = O(n \log n)$$

- **Worst Case:** Pivot is always the smallest or largest element:

$$T_{worst}(n) = O(n^2)$$

- **Average Case:** Random pivot gives balanced partitions:

$$T_{avg}(n) = O(n \log n)$$

- **Space Complexity:** Recursive stack depth:

$$S(n) = O(\log n)$$

Insertion Sort

Insertion Sort builds the sorted array one item at a time by inserting elements into their correct position.

Algorithm 3 Insertion Sort Algorithm

Require: Array $arr[0 \ldots n-1]$, integer n
Ensure: Sorted array arr

```
for i ← 1 to n − 1 do
    key ← arr[i]
    j ← i − 1
    while j ≥ 0 and arr[j] > key do
        arr[j + 1] ← arr[j]
        j ← j − 1
    end while
    arr[j + 1] ← key
end for
```

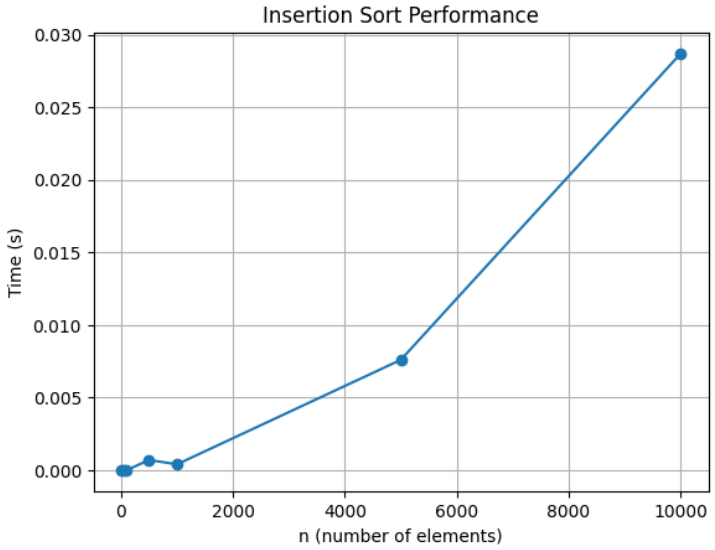
- Start from the second element.
- Compare with previous elements.
- Insert it into its correct position.

C Code:

```c
    #include <stdlib.h>
#include <stdio.h>
#include<time.h>
void insertionsort(int arr[],int n)
{
    for(int i=0;i<n;i++)
    {
         int key = arr[i];
         int j = i - 1;

     while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void generateRandomArray(int arr[],int n)
{
    for(int i=0;i<n;i++)
    {
        arr[i]=rand()%100000;//generate random integers
    }
}

```

```c
int main(int argc, char* argv[])
{
    if (argc < 2) {
        printf("Usage: %s <n>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);  // read n from command-line argument

    int* arr=(int*)malloc(n*sizeof(int));
    if(arr==NULL)
    {
        printf("Memory alloation failed!\n");
        return 1;
    }
    clock_t start=clock();
    for(int i=0;i<10;i++)
    {
        generateRandomArray(arr,n);
        insertionsort(arr,n);
    }
    clock_t end=clock();
    double time_taken=((double)(end-start))/CLOCKS_PER_SEC/10.0;//time for o
    printf("%f\n",time_taken);
    free(arr);
    return 0;
}
```

Graph:



Algorithmic Analysis

- **Best Case:** Already sorted array:

$$T_{best}(n) = O(n)$$

- **Worst Case:** Reverse sorted array:

$$T_{worst}(n) = O(n^2)$$

- **Average Case:** Random order array:

$$T_{avg}(n) = O(n^2)$$

- **Space Complexity:** Only requires constant space:

$$S(n) = O(1)$$

Selection Sort

Selection Sort repeatedly selects the minimum element from the unsorted portion and places it at the beginning.

Algorithm 4 Selection Sort Algorithm

Require: Array $arr[0 \ldots n-1]$, integer n
Ensure: Sorted array arr

```
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if arr[j] < arr[min] then
            min ← j
        end if
    end for
    Swap arr[i] and arr[min]
end for
```

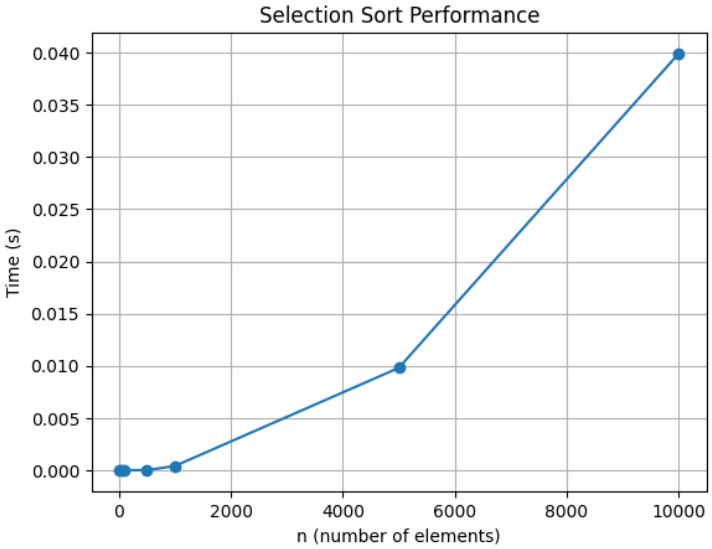
- Find the minimum element in the array.
- Swap it with the first element.
- Repeat for the remaining subarray.

C Code:

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

void selectionsort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {

        int min_idx = i;

        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[min_idx])
            {

                min_idx = j;
            }
        }
    }
}

void generateRandomArray(int arr[],int n)
{
    for(int i=0;i<n;i++)
```

```
26      {
27          arr[i]=rand()%100000;//generate random integers
28      }
29  }
30
31
32  int main(int argc, char* argv[])
33  {
34      if (argc < 2) {
35          printf("Usage: %s <n>\n", argv[0]);
36          return 1;
37      }
38
39      int n = atoi(argv[1]);  // read n from command-line argument
40      int* arr=(int*)malloc(n*sizeof(int));
41      if(arr==NULL)
42      {
43          printf("Memory alloation failed!\n");
44          return 1;
45      }
46      clock_t start=clock();
47      for(int i=0;i<10;i++)
48      {
49          generateRandomArray(arr,n);
50          selectionsort(arr,n);
51      }
52      clock_t end=clock();
53      double time_taken=((double)(end-start))/CLOCKS_PER_SEC/10.0;//time for o
54      printf(" %f\n",time_taken);
55      free(arr);
56
57      return 0;
58  }
```

Graph:



Algorithmic Analysis

- **Best Case:** Still scans entire array:

$$T_{best}(n) = O(n^2)$$

- **Worst Case:** Always n^2 comparisons:

$$T_{worst}(n) = O(n^2)$$

- **Average Case:** Same as worst:

$$T_{avg}(n) = O(n^2)$$

- **Space Complexity:** In-place sort:

$$S(n) = O(1)$$

Bubble Sort

Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order.

Algorithm 5 Bubble Sort Algorithm

Require: Array $arr[0 \ldots n-1]$, integer n
Ensure: Sorted array arr

```
for i ← 0 to n − 1 do
    for j ← 0 to n − i − 2 do
        if arr[j] > arr[j + 1] then
            Swap arr[j] and arr[j + 1]
        end if
    end for
end for
```

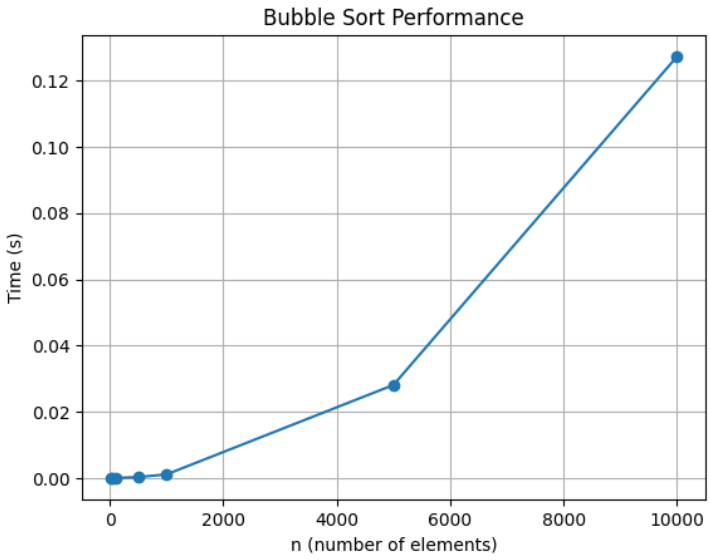
- Repeatedly compare adjacent elements.
- Swap them if they are in the wrong order.
- Continue until no swaps are needed.

C Code:

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

void swap(int* xp, int* yp){
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}


void bubblesort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
void generateRandomArray(int arr[],int n)
```

```
{
    for(int i=0;i<n;i++)
    {
        arr[i]=rand()%100000;//generate random integers
    }
}


int main(int argc, char* argv[])
{
    if (argc < 2) {
        printf("Usage: %s <n>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);  // read n from command-line argument
    int* arr=(int*)malloc(n*sizeof(int));
    if(arr==NULL)
    {
        printf("Memory alloation failed!\n");
        return 1;
    }
    clock_t start=clock();
    for(int i=0;i<10;i++)
    {
        generateRandomArray(arr,n);
        bubblesort(arr,n);
    }
    clock_t end=clock();
    double time_taken=((double)(end-start))/CLOCKS_PER_SEC/10.0;//time for o
    printf("%f\n",time_taken);
    free(arr);

    return 0;
}
```

Graph:



Algorithmic Analysis

- **Best Case:** Already sorted (with optimization):

$$T_{best}(n) = O(n)$$

- **Worst Case:** Reverse sorted array:

$$T_{worst}(n) = O(n^2)$$

- **Average Case:** Random order array:

$$T_{avg}(n) = O(n^2)$$

- **Space Complexity:** Only needs a few extra variables:

$$S(n) = O(1)$$