**Name: Aayush Basnet Date:**

**Lab 1: WAP to implement Bubble Sort.**

 **1. Theory:**

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until the list is sorted.

**2. Objective:**

The objective of this report is to:

* Understand the working principle of the Bubble Sort algorithm.
* Implement the algorithm in the C programming language.
* Analyze its performance empirically in terms of time complexity.

#### ****3. Algorithm****

The steps of Bubble Sort:

1. Start at the beginning of the list.
2. Compare each pair of adjacent elements.
3. Swap them if they are in the wrong order (ascending or descending).
4. Repeat steps 1–3 for all elements until the list is sorted.
5. Stop when no more swaps are needed.

**4. Code in C**

#include <stdio.h>

void bubbleSort(int arr[], int n) {

int i, j, temp;

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

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

if (arr[j] > arr[j + 1]) {

temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

int main() {

int n,i;

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

scanf("%d", &n);

int arr[n];

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

printf("Original array: ");

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

printf("%d ", arr[i]);

}

}

1. **Empirical Analysis**

For **Bubble Sort**, we analyze how the algorithm performs based on the number of elements (n) in the input array. Here's the analysis:

#### ****Time Complexity****

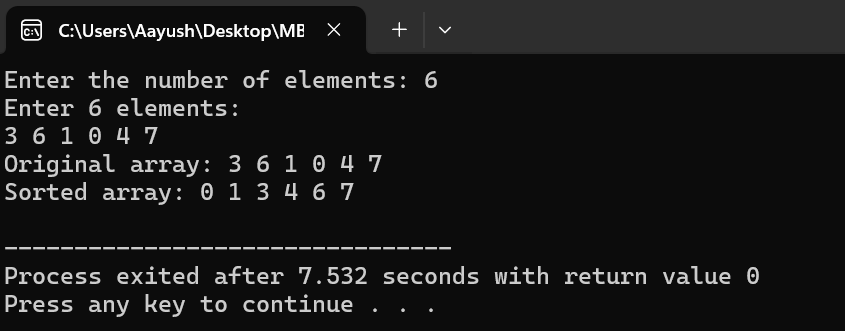
1. **Best Case (Already Sorted Array):**
   * When the array is already sorted, Bubble Sort requires a single pass (iteration) to confirm that no swaps are necessary.
   * Time Complexity: **O(n)** (linear).
2. **Worst Case (Reverse Sorted Array):**
   * The algorithm must compare every adjacent element and make the maximum number of swaps.
   * Time Complexity: **O(n²)** (quadratic).
3. **Average Case (Random Array):**
   * On average, Bubble Sort performs about half the total comparisons and swaps.
   * Time Complexity: **O(n²)** (quadratic).

#### ****Space Complexity****

* Bubble Sort operates directly on the input array (in-place sorting), so the space complexity is **O(1)** (constant auxiliary space).

### ****Conclusion****

* **Advantages**:
  + Simple and easy to implement.
  + Works well for small-sized arrays.
  + Stable sorting algorithm.
* **Disadvantages**:
  + Inefficient for large input sizes due to its **O(n²)** time complexity in the average and worst cases.
  + Not suitable for real-world large datasets where performance is critical.





**Name: Aayush Basnet**   **Date:**

**Lab 2: WAP to implement Fractional Knapsack.**

#### ****1. Theory****

The Fractional Knapsack Problem is an optimization problem that uses the **Greedy Algorithm** to maximize the total value of items that can be put into a knapsack with a given weight capacity.

#### ****2. Objective****

* To maximize the value of items placed in the knapsack while ensuring the total weight does not exceed the knapsack's capacity.
* To understand the application of the **Greedy Algorithm** in optimization problems.

#### ****3. Algorithm****

1. Calculate the value-to-weight ratio (value/weight) for each item.
2. Sort all items in descending order of their value-to-weight ratio.
3. Start with an empty knapsack and add as much as possible from the highest ratio item.
4. If the knapsack's capacity is exceeded, add a fraction of the current item.
5. Stop when the knapsack is full.

**Code in C**

#include <stdio.h>

struct Item {

int value, weight;

};

void fractionalKnapsack(struct Item items[], int n, int capacity) {

int i,j;

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

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

double r1 = (double)items[j].value / items[j].weight;

double r2 = (double)items[j + 1].value / items[j + 1].weight;

if (r1 < r2) {

struct Item temp = items[j];

items[j] = items[j + 1];

items[j + 1] = temp;

}

}}

double totalValue = 0.0;

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

if (capacity >= items[i].weight) {

// Take the full item

capacity -= items[i].weight;

totalValue += items[i].value;

} else {

// Take a fraction of the item

totalValue += items[i].value \* ((double)capacity / items[i].weight);

break;

}

}

printf("Maximum value in knapsack = %.2f\n", totalValue);

}

int main() {

int n,i, capacity;

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

scanf("%d", &n);

struct Item items[n];

printf("Enter value and weight of each item:\n");

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

printf("Item %d - Value: ", i + 1);

scanf("%d", &items[i].value);

printf("Item %d - Weight: ", i + 1);

scanf("%d", &items[i].weight);

}

printf("Enter the capacity of the knapsack: ");

scanf("%d", &capacity);

fractionalKnapsack(items, n, capacity);

return 0;

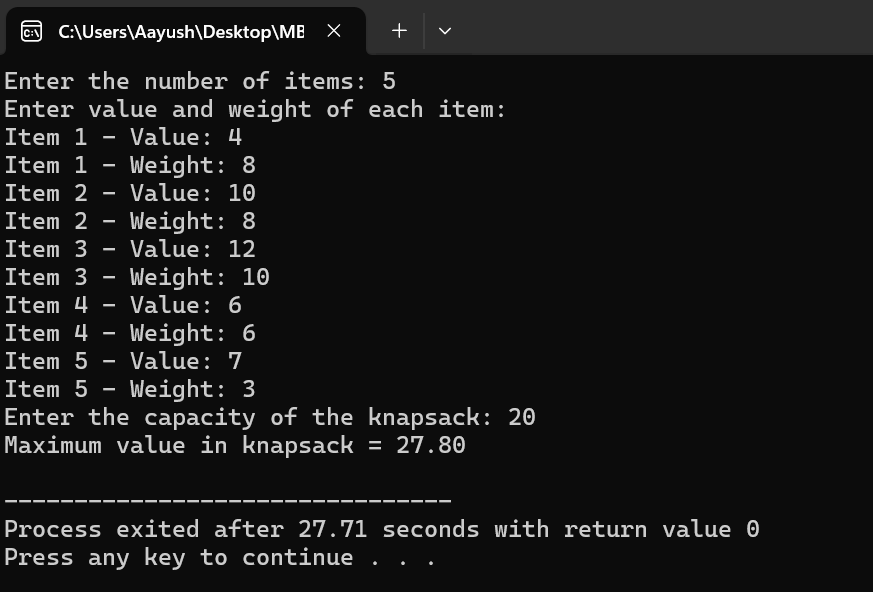
}

#### ****5. Empirical Analysis****

The time complexity of the Fractional Knapsack algorithm is determined by:

1. Sorting the items based on value-to-weight ratio: **O(n log n)**.
2. Iterating through the sorted list to fill the knapsack: **O(n)**.

* **Time Complexity**: **O(n log n)** (due to sorting).
* **Space Complexity**: **O(1)** (in-place sorting and processing).



#### ****6. Conclusion****

* **Advantages**:
  + Fractional Knapsack is efficient for scenarios where dividing items is possible (e.g., liquid goods).
  + Uses the **Greedy Algorithm**, ensuring optimal results.
* **Disadvantages**:
  + Not applicable when items cannot be divided (e.g., discrete goods like electronics).
  + Less practical in real-world scenarios compared to 0/1 Knapsack.



**Name: Aayush Basnet Date:**

**Lab 3: WAP to implement Heap Sort.**

#### ****1. Theory****

Heap Sort is a **comparison-based sorting algorithm** that uses a binary heap data structure to organize and sort data. A binary heap is a complete binary tree that satisfies the **heap property**:

* For a **max-heap**, the parent node is always greater than or equal to its child nodes.
* For a **min-heap**, the parent node is always less than or equal to its child nodes.

#### ****2. Objective****

* To implement and understand the **Heap Sort** algorithm.
* To efficiently sort data using a binary heap structure.

#### ****3. Algorithm****

1. Build a max-heap from the input array.
2. Swap the root (largest element) with the last element in the heap.
3. Reduce the size of the heap by one and heapify the root element.
4. Repeat steps 2 and 3 until the heap size is reduced to one.

**4. Code in C**

#include <stdio.h>

void heapify(int arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest])

largest = left;

if (right < n && arr[right] > arr[largest])

largest = right;

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n) {

int i;

for (i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

for (i = n - 1; i > 0; i--) {

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

}

void printArray(int arr[], int n) {

int i;

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

printf("%d ", arr[i]);

printf("\n");

}

int main() {

int n,i;

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

scanf("%d", &n);

int arr[n];

printf("Enter the elements:\n");

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

scanf("%d", &arr[i]);

}

printf("Original array: ");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

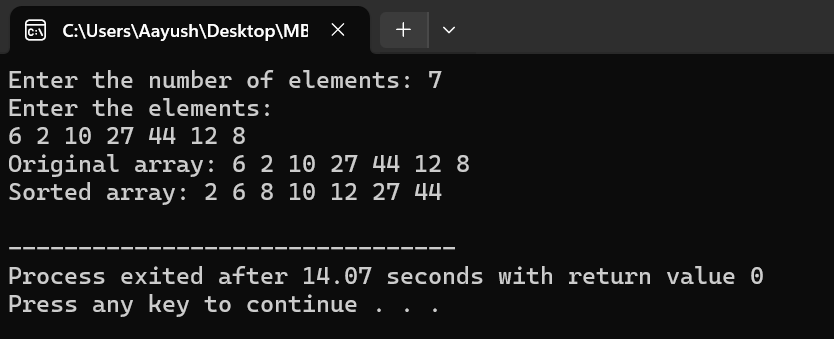
}

#### ****5. Empirical Analysis****

* **Time Complexity**:
  + Building a max-heap: **O(n)**
  + Heapify operation: **O(log n)** (performed for each element)
  + Sorting process: **O(n log n)**

**Overall Time Complexity**: **O(n log n)**

* **Space Complexity**: **O(1)** (in-place sorting with no extra space used).



#### ****6. Conclusion****

* **Advantages**:
  + Efficient with a guaranteed time complexity of **O(n log n)**.
  + Sorting is performed in-place, with no need for extra memory.
* **Disadvantages**:
  + Not stable (the relative order of equal elements may not be preserved).
  + Can be slower than other algorithms like Quick Sort in practice due to constant factors.



**Name: Aayush Basnet Date:**

**Lab 4: WAP to implement Merge Sort.**

#### ****1. Theory****

Merge Sort is a **divide-and-conquer** sorting algorithm. It recursively splits the input array into two halves until the size of the subarrays becomes one (base case), then merges the subarrays back in sorted order.

#### ****2. Objective****

* To understand the **Merge Sort** algorithm.
* To implement an efficient sorting algorithm that divides the problem into smaller subproblems and solves them recursively.

#### ****3. Algorithm****

1. **Divide**: Divide the array into two halves recursively until each subarray contains a single element.
2. **Conquer**: Merge the two halves in sorted order.
3. **Combine**: Repeat the merging process to produce a fully sorted array.

**4. Code in C**

/\* C program for Merge Sort \*/

#include<stdlib.h>

#include<stdio.h>

int count\_rec=0,count=0;

void merge(int arr[], int l, int m, int r) {

count++;

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

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

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

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

i = 0, j = 0, k = l;

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++;

}

}

void mergeSort(int arr[], int l, int r) {

count\_rec++;

if (l < r) {

int m = (r+l)/2;

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

void printArray(int A[], int size) {

int i;

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

printf("%d ", A[i]);

printf("\n");

printf("%d the total number of recursion\n",count\_rec);

printf("%d the total count to merge",count);

}

int main() {

int arr[] = {20, 8, 35, 7, 12, 12,34,46};

int arr\_size = sizeof(arr)/sizeof(arr[0]);

printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array is \n");

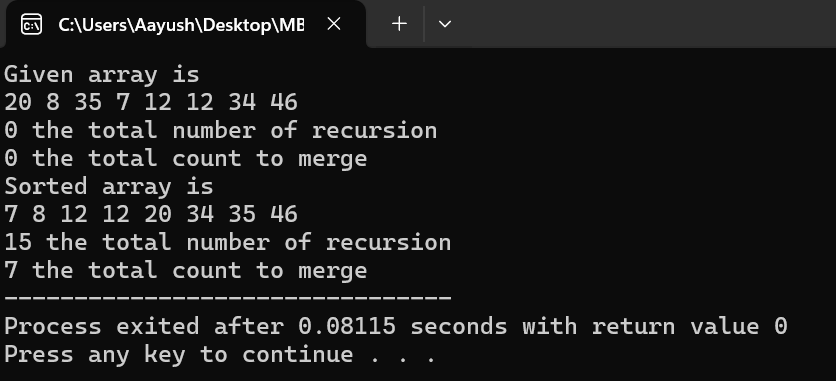
printArray(arr, arr\_size);

return 0;

}

#### ****5. Empirical Analysis****

* **Time Complexity**:
  + **Best Case**: **O(n log n)**
  + **Worst Case**: **O(n log n)**
  + **Average Case**: **O(n log n)**
* **Space Complexity**: **O(n)** due to the use of temporary arrays for merging.



#### ****6. Conclusion****

* **Advantages**:
  + Guaranteed time complexity of **O(n log n)** for all cases.
  + Stable sorting algorithm, preserving the relative order of equal elements.
  + Suitable for linked lists because random access is not required.
* **Disadvantages**:
  + Requires additional memory space for temporary arrays, which makes it unsuitable for environments with memory constraints.
  + Slower compared to in-place sorting algorithms like Quick Sort in practical scenarios.



**Name: Aayush Basnet Date:**

**Lab 5: WAP to implement Job Deadline Sequence.**

#### ****1. Theory****

The **Job Deadline Sequencing** problem is a classic optimization problem in computer science. The goal is to schedule a set of jobs such that the profit is maximized, given that each job has a deadline before which it must be completed. Each job takes a single unit of time to execute, and only one job can be executed at a time.

#### ****2. Objective****

* To determine the maximum profit that can be achieved by scheduling jobs within their deadlines.
* To implement the Job Sequencing algorithm to find the optimal solution.

#### ****3. Algorithm****

1. **Sort Jobs**: Arrange all the jobs in descending order of profit.
2. **Initialize Slots**: Create a slot array to keep track of free time slots up to the maximum deadline.
3. **Iterate Over Jobs**:
   * For each job, find the latest available slot before its deadline.
   * If a slot is available, assign the job to that slot.
4. **Calculate Profit**: Sum up the profits of the scheduled jobs.

**4. Code in C**

#include <stdio.h>

#define MAX 100

typedef struct Job {

char id[5];

int deadline;

int profit;

} Job;

void jobSequencingWithDeadline(Job jobs[], int n);

int minValue(int x, int y) {

if(x < y) return x;

return y;

}

int main(void) {

int i, j

Job jobs[5] = {

{"j1", 1, 60},

{"j2", 2, 100},

{"j3", 3, 20},

{"j4", 4, 40},

{"j5", 5, 20},

};

Job temp;

int n = 5;

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

for(j = 0; j < n - i; j++) {

if(jobs[j+1].profit > jobs[j].profit) {

temp = jobs[j+1];

jobs[j+1] = jobs[j];

jobs[j] = temp;

}

}

}

printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");

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

printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);

}

jobSequencingWithDeadline(jobs, n);

return 0;

}

void jobSequencingWithDeadline(Job jobs[], int n) {

int i, j, k, maxprofit, count = 0;

int timeslot[MAX], filledTimeSlot = 0;

int dmax = 0;

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

if(jobs[i].deadline > dmax) {

dmax = jobs[i].deadline;

}

}

for(i = 1; i <= dmax; i++) {

timeslot[i] = -1;

}

printf("dmax: %d\n", dmax);

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

count++;

k = minValue(dmax, jobs[i - 1].deadline);

while(k >= 1) {

count++;

if(timeslot[k] == -1) {

timeslot[k] = i-1;

filledTimeSlot++;

break;

}

k--;

}

if(filledTimeSlot == dmax) {

break;

}

}

printf("\nRequired Jobs: ");

for(i = 1; i <= dmax; i++) {

printf("%s", jobs[timeslot[i]].id);

if(i < dmax) {

printf(" --> ");

}

}

maxprofit = 0;

for(i = 1; i <= dmax; i++) {

maxprofit += jobs[timeslot[i]].profit;

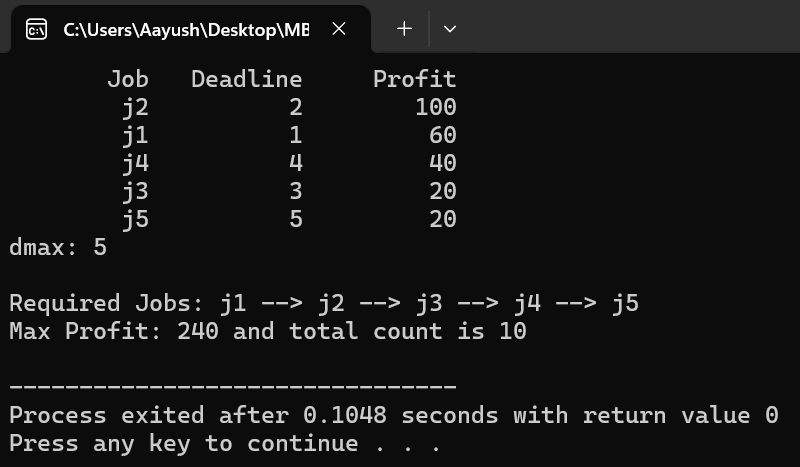
}

printf("\nMax Profit: %d and total count is %d\n", maxprofit,count);

}

#### ****5. Empirical Analysis****

* **Time Complexity**:
  + Sorting jobs by profit: **O(n log n)**.
  + Allocating slots for each job: **O(n × d)**, where d is the maximum deadline.
  + Total time complexity: **O(n log n + n × d)**.
* **Space Complexity**: **O(d)**, where d is the maximum deadline (for the slots array).



#### ****6. Conclusion****

* **Advantages**:
  + Greedy approach ensures the solution is close to optimal in many real-world cases.
  + Easy to implement and efficient for small to medium-sized datasets.
* **Disadvantages**:
  + May not work efficiently for very large datasets due to the time complexity involving n × d.
  + Assumes all jobs take exactly one unit of time, which may not be realistic in some scenarios.



**Name: Aayush Basnet Date:**

**Lab 6: WAP to implement Selection Sort.**

#### ****1. Theory****

Selection Sort is a sorting algorithm that works by repeatedly finding the minimum element from the unsorted portion of the array and placing it at the beginning. The process involves two subarrays:

1. The sorted subarray.
2. The unsorted subarray.

#### ****2. Objective****

* To sort an array in ascending or descending order using the **Selection Sort** algorithm.
* To demonstrate its working and evaluate its performance.

#### ****3. Algorithm****

1. **Input**: An array of size nnn.
2. **Procedure**:
   * Start from the first element.
   * Find the smallest element in the unsorted portion of the array.
   * Swap it with the leftmost unsorted element.
   * Repeat for the rest of the array.
3. **Output**: The array is sorted in ascending or descending order.
4. **Code in C**

#include <stdio.h>

int array[50],i,j,temp,n,min=0,count=0;

int main(){

printf("Enter the size: ");

scanf("%d",&n);

printf("Enter the elements: ");

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

scanf("%d",&array[i]);

}

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

min=i;

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

count++;

if(array[j]<array[min]){

min=j;

}

}

temp=array[i];

array[i]=array[min];

array[min]=temp;

}

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

printf("%d\t",array[i]);

}

printf("%d",count);

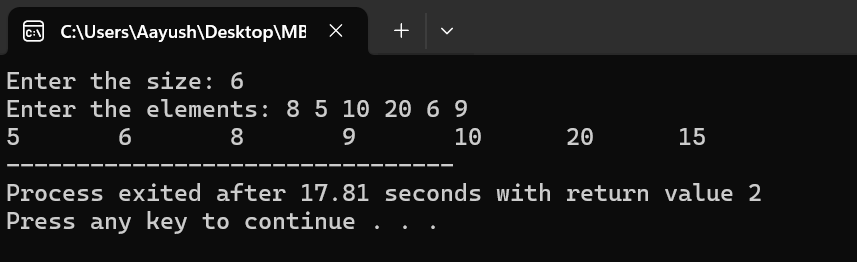
}

#### ****5. Empirical Analysis****

* **Time Complexity**:
  + Best Case: O(n^2) (array is already sorted).
  + Average Case: O(n^2).
  + Worst Case: O(n^2) (array is sorted in reverse order).
* **Space Complexity**:

O(1), as it is an in-place sorting algorithm.

* **Stability**:
  + Selection Sort is **not stable** because it may swap non-adjacent elements.

****

#### ****6. Conclusion****

* **Advantages**:
  + Simple and easy to implement.
  + Suitable for small datasets.
  + Does not require additional memory.
* **Disadvantages**:
  + Inefficient for large datasets due to O(n^2) complexity.
  + Not a stable sort, which can be an issue in some cases.



**Name: Aayush Basnet Date:**

**Lab 7: WAP to implement Order Statistics.**

#### ****1. Theory****

In computer science, **order statistics** refer to the elements of a dataset arranged in a particular order. Specifically, the kth order statistic of a set of n elements is the kth smallest element in the sorted order of the dataset.

For example:

* **Minimum Element**: 1st order statistic.
* **Maximum Element**: nth order statistic.
* **Median**:
  + If n is odd, it is the (n+1)/2th order statistic.
  + If n is even, it is the average of (n/2)th and (n/2+1)th order statistics.

#### ****Objective****

* To determine the kth smallest or largest element in an array.
* To provide an efficient algorithm for finding order statistics without fully sorting the array.

#### ****3. Algorithm****

**Naive Approach**:

1. Sort the array in ascending order.
2. Return the kth element.

**Efficient Approach** (Quickselect):

1. Use the **Quickselect** algorithm, a variant of Quicksort.
2. Partition the array around a pivot.
3. Recursively narrow down the search to the partition that contains the kthk^{th}kth smallest element.

**Code in C**

#include <stdio.h>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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);

}

int quickselect(int arr[], int low, int high, int k) {

if (low <= high) {

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

if (pivotIndex == k - 1) {

return arr[pivotIndex];

} else if (pivotIndex > k - 1) {

return quickselect(arr, low, pivotIndex - 1, k);

} else {

return quickselect(arr, pivotIndex + 1, high, k);

}

}

return -1;

}

int main() {

int n, k;

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

scanf("%d", &n);

int arr[n];

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

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

scanf("%d", &arr[i]);

}

printf("Enter the value of k: ");

scanf("%d", &k);

int result = quickselect(arr, 0, n - 1, k);

if (result != -1) {

printf("The %d-th smallest element is: %d\n", k, result);

} else {

printf("Invalid value of k\n");

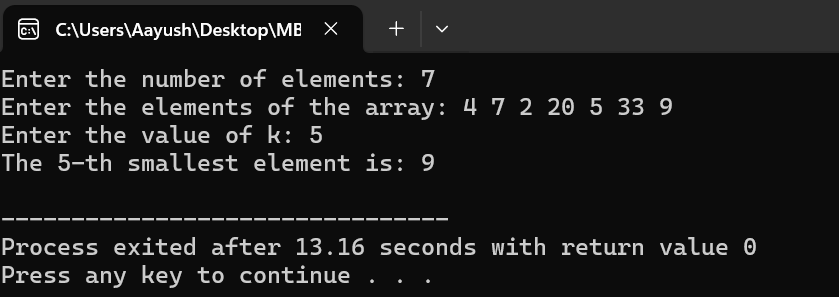
}

return 0;

}

#### ****5. Empirical Analysis****

* **Time Complexity**:
  + Best Case: O(n) (when the pivot divides the array into equal halves).
  + Average Case: O(n).
  + Worst Case: O(n2) (when the pivot is the smallest or largest element repeatedly).
* **Space Complexity**: O(1), as Quickselect is an in-place algorithm.



#### ****6. Conclusion****

* **Advantages**:
  + Quickselect is more efficient than sorting for finding order statistics in large datasets.
  + The algorithm is simple and operates in O(n) average time.
* **Disadvantages**:
  + The worst-case time complexity is O(n2).
  + Not suitable when all elements need to be sorted, as sorting algorithms like Merge Sort or Quick Sort are better suited.



**Name: Aayush Basnet Date:**

**Lab 8: WAP to implement Factorial.**

#### ****1. Theory****

The **factorial** of a non-negative integer n, denoted as n!, is the product of all positive integers less than or equal to n. It is defined as:

n! =n×(n−1)×(n−2)×⋯× 1

For n = 0, the factorial is defined as 0! = 1 (by convention).

Examples:

* 5! = 5×4×3×2×1= 120
* 3! = 3×2×1=6

#### ****2. Objective****

* To compute the factorial of a number nnn using iterative and recursive approaches.
* To understand the mathematical and computational efficiency of factorial calculations.

#### ****3. Algorithm****

##### **Iterative Approach**:

1. Initialize a variable fact to 1.
2. Multiply fact by every integer from 1 to n.
3. Return the value of fact.

##### **Recursive Approach**:

1. Base Case:
   * If n = 0 or n=1, return 1.
2. Recursive Case:
   * Return n × factorial(n−1)

**Code in C**

#include<stdio.h>

#include<conio.h>

int fact(int n);

int count = 0;

int main(){

int n,retval;

printf("Enter the number to get factorial: ");

scanf("%d",&n);

retval = fact(n);

printf("fact is %d \nRate of growth for %d is %d",retval,n,count);

return 0;

}

int fact(int n){

count++;

if(n==0)

return 1;

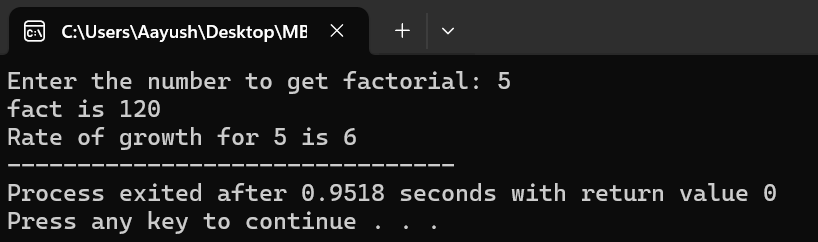
else

return n \* fact(n-1);

}

#### ****5. Empirical Analysis****

* **Time Complexity**:
  + Iterative Approach: O(n)
  + Recursive Approach: O(n)
* **Space Complexity**:
  + Iterative Approach: O(1) (constant space used).
  + Recursive Approach: O(n) (stack space due to recursion).



#### ****6. Conclusion****

* **Advantages**:
  + Factorials are essential in combinatorics, probability, and various mathematical fields.
  + Both iterative and recursive methods are straightforward to implement.
* **Disadvantages**:
  + Recursive implementations may result in **stack overflow** for very large numbers.
  + Factorials grow rapidly, and calculations can exceed standard data type limits for large n.



**Name: Aayush Basnet Date:**

**Lab 9: WAP to implement Fibonacci Numbers.**

**1. Theory**

The **Fibonacci series** is a sequence of numbers where each number is the sum of the two preceding ones. It starts with 0 and 1 as the first two terms.

The series looks like this:

0,1,1,2,3,5,8,13,21,34,….

#### ****2. Objective****

* To calculate and display the Fibonacci series up to n-terms.
* To understand the recursive and iterative approaches to generating the Fibonacci sequence.

#### ****3. Algorithm****

##### **Iterative Approach**:

1. Initialize the first two Fibonacci numbers: F(0) = 0 and F(1)=1.
2. Loop from 2 to n−1, and calculate the next Fibonacci number as the sum of the previous two numbers.
3. Store and display the Fibonacci numbers.

##### **Recursive Approach**:

1. Base Case:
   * If n = 0, return 0.
   * If n = 1, return 1.
2. Recursive Case:
   * Return F(n−1) + F(n−2)

**Code in C**

#include<stdio.h>

#include<conio.h>

int fib(int n);

int count = 0;

int main()

{

int n,retval;

printf("Enter the number to get fibonacci: ");

scanf("%d",&n);

retval = fib(n);

printf("fib is %d \nRate of growth for %d is %d",retval,n,count);

return 0;

}

int fib(int n){

if(n<=1){

count++;

return n;

} else{

count++;

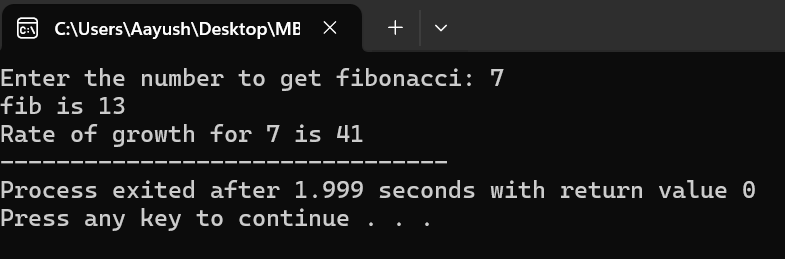
return fib(n-1) + fib(n-2);

}

}

#### ****5. Empirical Analysis****

* **Time Complexity**:
  + Iterative Approach: O(n)
  + Recursive Approach: O(2^n) (due to redundant calculations in naive recursion).
* **Space Complexity**:
  + Iterative Approach: O(1)
  + Recursive Approach: O(n) (stack space due to recursion).



#### ****6. Conclusion****

* **Advantages**:
  + Fibonacci series has a simple and straightforward implementation.
  + It is widely used in mathematical computations, algorithms, and computer science applications.
* **Disadvantages**:
  + Recursive implementations can be inefficient for large nnn, due to exponential growth in function calls.
  + Fibonacci numbers grow rapidly, requiring larger data types for higher terms.