CCN UNIVERSITY
OF SCIENCE & TECHNOLOGY

Lab Report

Course Title: Operating System (Sessional)
Course Code: CSE-132 (Sessional)

Submission Date: 13-05-2024

Submitted to,	**Submitted by,**
Farhana Abedin Lecturer Dept. of CSE CCNUST	Kamrul Hasan 111221022 CSE 8th Batch CCNUST

Bubble Sort

Introduction:

Bubble sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. It is named for the way smaller or larger elements "bubble" to the top of the list with each iteration. While not the most efficient sorting algorithm, bubble sort is easy to understand and implement, making it a common choice for educational purposes and small datasets.

Code:

```
#include <stdio.h>


void merge(int arr[], int l, int m, int r) {
    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) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
```

```
        merge(arr, l, m, r);
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Input & Output:

Advantages:

While bubble sort is not the most efficient sorting algorithm, it does have some advantages:

1.Simple Implementation: Bubble sort is one of the simplest sorting algorithms to implement. It requires only basic looping and swapping operations, making it easy to understand and implement, especially for beginners.

2.No Extra Memory Requirement: Bubble sort operates directly on the input array without requiring additional memory space. This can be advantageous in scenarios where memory usage is a concern.

3.Adaptive: Bubble sort can be adaptive in certain cases, meaning that it can take advantage of partially sorted arrays by terminating early if no swaps are performed during a pass. This can lead to improved performance on nearly sorted lists.

4.Stable Sorting: Bubble sort is a stable sorting algorithm, meaning that the relative order of equal elements is preserved during sorting. This property can be important in applications where maintaining the original order of equal elements is necessary.

5.Useful for Small Data Sets: While bubble sort is not efficient for large datasets, it can be suitable for sorting small arrays or lists where efficiency is less of a concern. In such cases, its simplicity and ease of implementation may outweigh its relatively slower performance.

Disadvantages:

Here are five disadvantages of bubble sort:

1.Inefficient for Large Datasets: Bubble sort has a time complexity of O(n^2), making it highly inefficient for sorting large datasets. As the size of the input array increases, the number of comparisons and swaps also grows quadratically, leading to poor performance compared to more efficient sorting algorithms.

2.Poor Performance on Nearly Sorted Data: While bubble sort can be adaptive in some cases, it still requires multiple passes through the entire list to fully sort the data. This makes it inefficient for nearly sorted or partially sorted arrays, as it does not take full advantage of the pre-existing order.

3.Lack of Practicality: Bubble sort's inefficiency makes it impractical for real-world applications where sorting large datasets quickly is essential. Other sorting algorithms such as quicksort, mergesort, or heapsort offer much better performance in such scenarios.

4.Not Suitable for Large-scale Sorting: Due to its poor time complexity, bubble sort is not suitable for sorting large-scale datasets commonly encountered in databases, web applications, or data processing tasks. Using bubble sort on such datasets can lead to significant processing delays and decreased overall system performance.

5.Low Performance on Reverse-ordered Data: Bubble sort performs poorly on reverse-ordered or nearly reverse-ordered data. In such cases, it requires the maximum number of comparisons and swaps, leading to its worst-case time complexity scenario.

Insertion sort

Introduction:

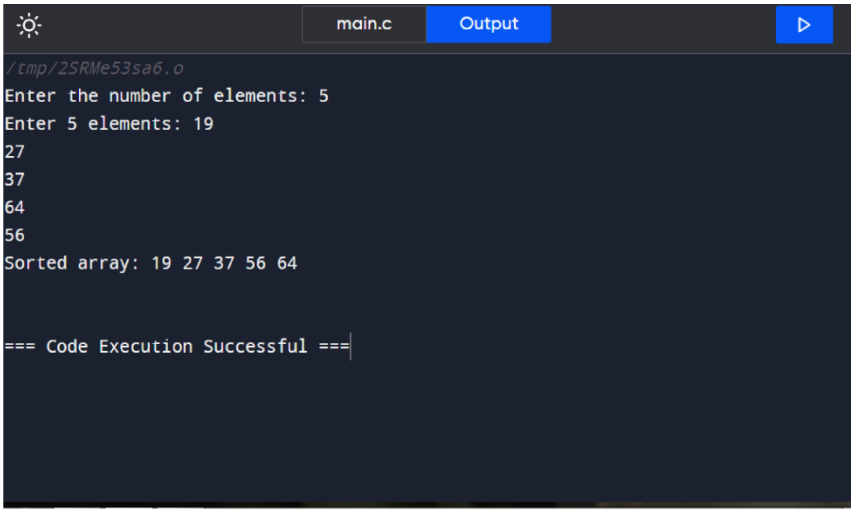
Insertion sort is a simple comparison-based sorting algorithm that builds the final sorted array or list one element at a time. It iterates through the input array, removing one element at a time and inserting it into the correct position in the sorted portion of the array. At each iteration, the algorithm selects the current element and compares it with the elements in the sorted subarray, shifting larger elements one position to the right until the correct position for the current element is found. Insertion sort has a time complexity of O(n^2) in the worst case but performs well on small datasets or nearly sorted arrays. It is often used in practice for sorting small arrays or as a subroutine in more complex sorting algorithms.

Code:

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    bubbleSort(arr, n);
```

```
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Input & Output:



Advantages:

Here are five advantages of insertion sort:

1.Efficiency for Small Data Sets: Insertion sort performs well on small datasets or lists. Its simple implementation and low overhead make it efficient for sorting small collections of elements, often outperforming more complex algorithms on such datasets.

2.Adaptive: Insertion sort is adaptive, meaning that it performs well on nearly sorted arrays. In scenarios where the input data is partially sorted or already sorted to some extent, insertion sort requires fewer comparisons and swaps, leading to improved performance.

3.In-place Sorting: Insertion sort can be implemented as an in-place sorting algorithm, meaning it does not require additional memory space proportional to the size of the input array. This makes it suitable for sorting large datasets with limited memory resources.

4.Stable Sorting: Insertion sort is a stable sorting algorithm, meaning that the relative order of equal elements is preserved during sorting. This stability can be important in applications where maintaining the original order of equal elements is necessary.

5.Simple Implementation: Insertion sort has a straightforward implementation that is easy to understand and implement. It involves iterating through the input array and inserting each element into its correct position in the sorted subarray, making it accessible to programmers of all levels.

Disadvantages:

Here are five disadvantages of insertion sort:

1.Quadratic Time Complexity: Insertion sort has a time complexity of O(n^2) in the worst-case scenario, where n is the number of elements in the array. This makes it inefficient for sorting large datasets, as the number of comparisons and swaps increases quadratically with the size of the input.

2.Inefficiency for Large Data Sets: Due to its quadratic time complexity, insertion sort becomes inefficient for sorting large datasets. As the size of the input array grows, insertion sort's performance degrades rapidly compared to more efficient sorting algorithms like quicksort or mergesort.

3.Non-Adaptivity: Insertion sort is non-adaptive, meaning that it does not change its behavior based on the characteristics of the input data. It performs the same number of comparisons and swaps regardless of whether the data is already partially sorted or completely unsorted.

4.Not Suitable for Reverse-Ordered Data: Insertion sort performs poorly on reverse-ordered or nearly reverse-ordered data. In such cases, it requires the maximum number of comparisons and swaps, leading to its worst-case time complexity scenario.

5.**Limited Use in Practice:** While insertion sort is efficient for small datasets or nearly sorted arrays, it is rarely used in practice for sorting large datasets due to its inefficiency. Other sorting algorithms such as quicksort, mergesort, or heapsort offer better performance and scalability for most real-world sorting tasks.

Binary search

Introduction:

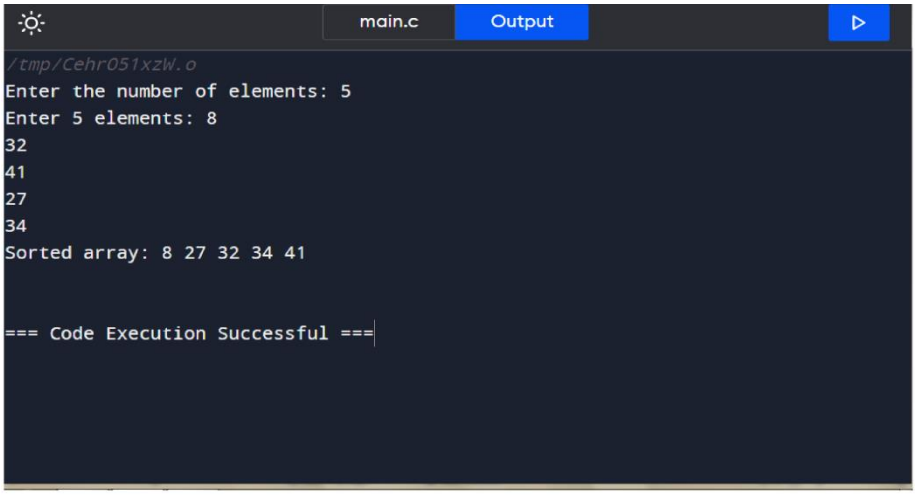
Binary search is a fundamental algorithm used for searching in sorted arrays or lists by repeatedly dividing the search interval in half. It efficiently locates a target value within the data set by comparing the target with the middle element of the array and adjusting the search range accordingly. This approach significantly reduces the number of elements to be searched, making it a fast and efficient method for finding elements in large datasets.

Code:

```
#include <stdio.h>


void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}


int main() {
    int n;
```

```
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Input & Output:



Advantages:

Here are five advantages of binary search:

1.Efficiency: Binary search has a time complexity of O(log n), making it highly efficient, especially for large datasets. It significantly reduces the number of elements to be searched compared to linear search algorithms.

2.Versatility: Binary search can be applied to various types of sorted data structures, including arrays, linked lists, and trees. This versatility makes it a valuable algorithm in many programming contexts.

3.Ease of Implementatio: The binary search algorithm is relatively simple to implement and understand, requiring only a few lines of code. This simplicity makes it accessible to programmers of all levels.

4.Optimal Performance: In scenarios where the data is sorted and static, binary search provides optimal performance. Once the data is sorted, binary search consistently delivers fast search times, regardless of the size of the dataset.

5. Deterministic Behavior: Binary search exhibits deterministic behavior, meaning that it always produces the same result for a given input dataset. This predictability makes it a reliable choice for search operations in applications where consistency is crucial.

Disadvantages:

Here are five disadvantages of binary search:

1.Requirement of Sorted Data: Binary search requires the data to be sorted beforehand. If the data is not sorted, additional preprocessing steps are needed to ensure the data meets this requirement, which can increase both time and space complexity.

2.Limited Applicability: Binary search can only be used on sorted data structures such as arrays, trees, or lists. This limitation restricts its applicability in scenarios where sorting the data is not feasible or where the data structure cannot be sorted efficiently.

3.Not Suitable for Dynamic Dat: Binary search is not suitable for dynamic datasets where elements are frequently added or removed. Each insertion or deletion may require reordering the data, which can be inefficient and negate the benefits of binary search.

4.Memory Overhead for Recursive Implementation: Recursive implementations of binary search may incur additional memory overhead due to the function call stack. In scenarios with deeply nested recursive calls, this overhead can lead to stack overflow errors, especially for large datasets.

5. Inability to Handle Duplicates: Binary search may not handle duplicate elements well, as it may return any index where the target value is found. This behavior may not be desirable in applications where precise handling of duplicates is required.

Merge Sort

Introduction:

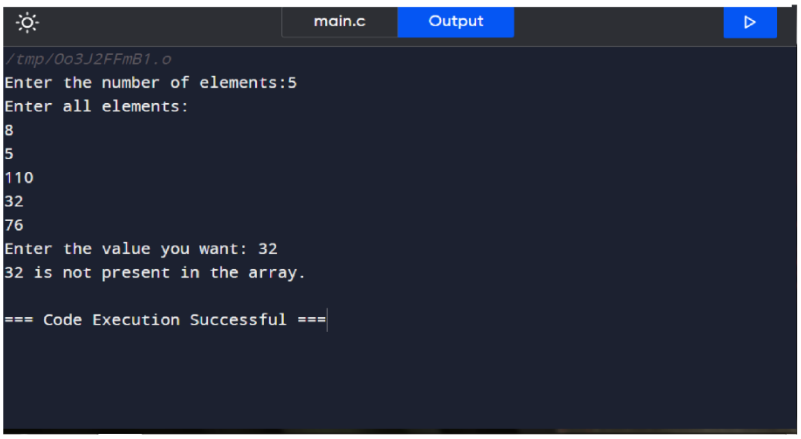
Merge sort is a divide-and-conquer algorithm used for sorting elements in an array or list. It works by recursively dividing the input array into smaller subarrays until each subarray contains only one element. Then, it merges these smaller sorted subarrays back together to produce a single sorted array. Merge sort has a time complexity of O(n log n), making it efficient for large datasets, and it is known for its stable sorting behavior, meaning that the relative order of equal elements is preserved during sorting. Due to its efficiency and stability, merge sort is widely used in various applications where sorting is required.

Code:

```
#include<stdio.h>
int main()
{
        int n,i,a,low,high,mid;

 printf("Enter the number of elements:");
 scanf("%d", &n);

 int array[n];
 printf("Enter all elements:\n");

for(i=0;i<n;i++)
        scanf("%d", &array[i]);
printf("Enter the value you want: ");
        scanf("%d", &a);

low=0;
high= n-1;
mid=(low+high)/2;

while(low<=high)
{
 if(array[mid]<a)
    low=mid+1;

 else if(array[mid]==a)
 {
```

```
        printf("%d found at the location %d.",a,mid+1);
break;
}
else
        high=mid-1;
   mid=(low+high)/2;
 }
 if(low>high)
        printf("%d is not present in the array.", a);


    return 0;
 }
```

Input & Output:



Advantages:

Here are five advantages of merge sort:

1.Efficienc: Merge sort has a time complexity of O(n log n), making it highly efficient for sorting large datasets. Its efficient divide-and-conquer approach ensures that the sorting time remains reasonable even as the dataset size increases.

2.Stable Sorting: Merge sort is a stable sorting algorithm, meaning that the relative order of equal elements is preserved during sorting. This stability can be crucial in applications where maintaining the original order of equal elements is important.

3.Adaptability: Merge sort performs consistently well across different types of input data, including arrays with various degrees of pre-sorting. It does not suffer from performance degradation on nearly sorted or partially sorted datasets, unlike some other sorting algorithms.

4.Parallelizability: Merge sort is easily parallelizable, as its divide-and-conquer nature allows for efficient parallel processing of subproblems. This makes it suitable for implementations in parallel computing environments, where multiple processors or threads can work on different parts of the sorting process simultaneously.

5.Widely User: Merge sort is widely used in various applications and programming languages due to its efficiency, stability, and ease of implementation. Many standard libraries and sorting routines implement merge sort or variants of it, making it a trusted choice for sorting tasks.

Disadvantages:

Here are five disadvantages of merge sort:

1.Space Complexity: Merge sort requires additional space proportional to the size of the input array for the temporary arrays used during the merging phase. This additional space requirement can be a disadvantage in memory-constrained environments or when sorting very large datasets.

2. Not In-place: Merge sort is not an in-place sorting algorithm, meaning it requires additional memory space to store the sorted elements. This can be a disadvantage when memory usage is a concern, especially for large datasets.

3. Complexity in Linked Lists: Merge sort can be less efficient when sorting linked lists compared to arrays. While it still maintains its O(n log n) time complexity, the additional memory overhead and pointer manipulation required for merging can make it less practical for linked lists.

4.Not Adaptive: Merge sort does not adapt its behavior based on the characteristics of the input data. It always divides the input array into halves, regardless of whether the data is already partially sorted. This lack of adaptability can result in unnecessary work when sorting partially sorted datasets.

5.Recursive Overhead: Merge sort's recursive nature can lead to stack overflow errors when sorting very large arrays, especially in environments with limited stack space. While this can be mitigated by

using iterative versions of merge sort or increasing the stack size, it remains a potential disadvantage in certain scenarios.

First Come First Serve (FCFS)

Introduction:

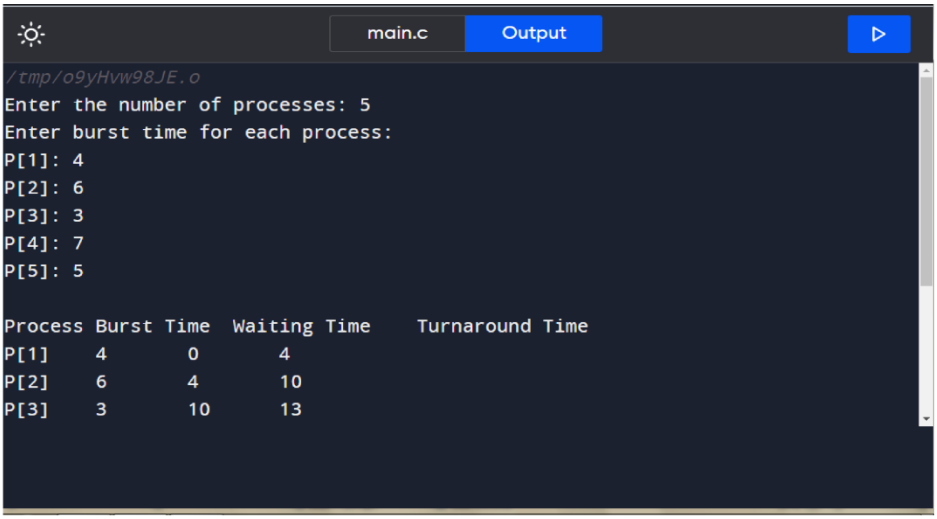
FCFS stands for First-Come, First-Served, and it's one of the simplest scheduling algorithms used in computing. In FCFS, processes are executed in the order they arrive in the system's ready queue. The process that arrives first is executed first, followed by the next process in the queue, and so on. FCFS is easy to implement and understand, but it may not always be the most efficient scheduling algorithm, especially in scenarios where shorter jobs arrive later, leading to longer average waiting times. Despite its simplicity, FCFS is still used in certain contexts, such as batch processing or environments where fairness in scheduling is prioritized over efficiency.

Code:

```
#include<stdio.h>


int main() {
    int n, i;
    float avg_waiting_time = 0, avg_turnaround_time = 0;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burst_time[n], waiting_time[n], turnaround_time[n];
    printf("Enter burst time for each process:\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]: ", i+1);
        scanf("%d", &burst_time[i]);
    }
```

```
    waiting_time[0] = 0;
    for(i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i-1] + burst_time[i-1];
        avg_waiting_time += waiting_time[i];
    }
    for(i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + burst_time[i];
        avg_turnaround_time += turnaround_time[i];
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]\t%d\t\t%d\t\t%d\n", i+1, burst_time[i], waiting_time[i],
turnaround_time[i]);
    }
    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    return 0;
}
```

Input & Output:



Advantages:

Here are five advantages of the FCFS (First-Come, First-Served) scheduling algorithm:

1.Simple Implementation: FCFS is straightforward to implement, making it easy for operating systems to manage and for programmers to understand. It involves executing processes in the order they arrive, without any complex logic or decision-making.

2.Fairness: FCFS ensures fairness by treating all processes equally in terms of execution order. This fairness principle can be desirable in scenarios where equal access to resources is prioritized over optimization of resource utilization or response time.

3.No Starvation: FCFS prevents starvation, as it guarantees that every process eventually gets a chance to execute. Since processes are scheduled based on their arrival time, no process is indefinitely delayed or overlooked, regardless of its priority.

4.No Overhead: FCFS scheduling does not incur any additional overhead or computational complexity beyond maintaining a simple queue of processes. This lack of overhead makes FCFS suitable for systems with limited computational resources or where simplicity is preferred over optimization.

5.Low Context Switching: FCFS minimizes context switching overhead because it does not require frequent switching between processes. Once a process starts executing, it continues until completion, reducing the overhead associated with saving and restoring process states.

Disadvantages:

Here are five disadvantages of the FCFS (First-Come, First-Served) scheduling algorithm:

1.Poor Turnaround Time: FCFS can lead to poor turnaround times, especially when long processes arrive first, causing shorter processes to wait for extended periods before execution. This can result in inefficient resource utilization and increased average waiting times.

2.Convoy Effect: FCFS can suffer from the convoy effect, where shorter processes get stuck behind a long-running process. This can create a convoy of processes waiting for the long process to finish, leading to increased waiting times and reduced system throughput.

3.Inefficient Resource Utilization: FCFS may result in inefficient resource utilization, as it does not consider the length of processes or the availability of resources. Shorter processes may be delayed unnecessarily if they arrive after longer processes, leading to suboptimal use of CPU and other system resources.

4.No Prioritization: FCFS does not prioritize processes based on their importance or urgency. Critical or time-sensitive processes may be delayed if they arrive after less critical processes, which can be detrimental in real-time or high-priority computing environments.

5.Vulnerability to Blocking: FCFS is vulnerable to blocking situations where a process waits for an I/O operation to complete before it can proceed. If a long-running I/O-bound process arrives early, it may block subsequent CPU-bound processes from executing, leading to increased waiting times and reduced system responsiveness.

Shortest Job First (SJF)

Introduction:

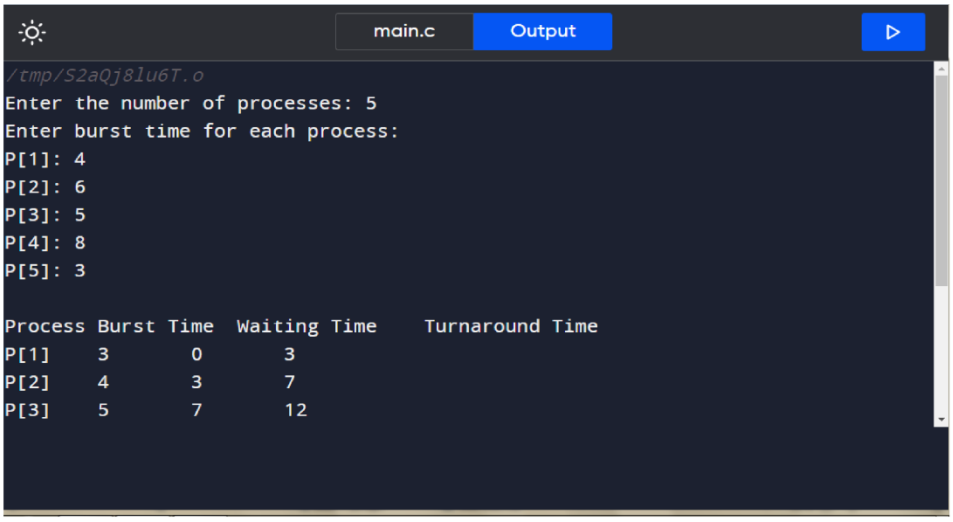
Shortest Job First (SJF) is a CPU scheduling algorithm that selects the waiting process with the smallest execution time to execute next. It's a non-preemptive scheduling algorithm, meaning once a process starts executing, it continues until it finishes or blocks. SJF minimizes the average waiting time for all processes, making it one of the most efficient scheduling algorithms, especially for batch systems where all processes are available upfront. Let me know if you need more information on SJF!

Code:

```
#include<stdio.h>


int main() {
    int n, i, j, temp;
    float avg_waiting_time = 0, avg_turnaround_time = 0;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burst_time[n], waiting_time[n], turnaround_time[n];
    printf("Enter burst time for each process:\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]: ", i+1);
        scanf("%d", &burst_time[i]);
    }
    for(i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
            if(burst_time[j] > burst_time[j+1]) {
                temp = burst_time[j];
                burst_time[j] = burst_time[j+1];
                burst_time[j+1] = temp;
            }
        }
```

```
    }
    waiting_time[0] = 0;
    for(i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i-1] + burst_time[i-1];
        avg_waiting_time += waiting_time[i];
    }
    for(i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + burst_time[i];
        avg_turnaround_time += turnaround_time[i];
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]\t%d\t\t%d\t\t%d\n", i+1, burst_time[i], waiting_time[i],
turnaround_time[i]);
    }
    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    return 0;
}
```

Input & Output:



Advantages:

Here are five advantages of Shortest Job First (SJF) scheduling algorithm:

1.Minimized Average Waiting Time: SJF aims to execute the shortest jobs first, which leads to reduced waiting times for processes overall. This results in better system throughput and responsiveness.

2.Optimal for Short Jobs: SJF is particularly effective when most of the processes are short-lived. By prioritizing short jobs, it ensures that they get completed quickly, leading to higher system efficiency.

3.Improved Turnaround Time: Since SJF prioritizes shorter jobs, it tends to minimize the turnaround time, which is the total time taken to execute a process from submission to completion.

4.Fairness: SJF provides fairness by giving priority to shorter jobs. This prevents long-running processes from hogging the CPU, ensuring that all processes get executed in a reasonable time frame.

5.Efficient Resource Utilization: By executing shorter jobs first, SJF optimizes CPU utilization. It reduces the likelihood of longer processes monopolizing the CPU, allowing for better overall resource utilization and system performance. These advantages make SJF a preferred choice in certain scenarios where the characteristics of the workload align with its strengths.

Disadvantages:

Here are five disadvantages of the Shortest Job First (SJF) scheduling algorithm:

1.Predicting Job Length: SJF requires accurate estimation or prediction of the execution time for each process. In real-world scenarios, it's often challenging to accurately determine the duration of a process before it starts executing, leading to potential inefficiencies.

2.Starvation of Longer Jobs: Longer jobs or processes with higher execution times may suffer from starvation in SJF, as shorter jobs continuously get priority. This can result in longer processes waiting indefinitely for execution, leading to poor system performance.

3. Preemption Overhead: SJF is typically implemented as a non-preemptive scheduling algorithm, meaning once a process starts execution, it runs to completion. However, in scenarios where preemption is allowed (such as Shortest Remaining Time First, a variant of SJF), there can be overhead associated with context switching and managing preemption.

4.Dependency on Process Arrival Time: SJF's effectiveness can be impacted by the order in which processes arrive. If longer jobs arrive before shorter ones, they might face significant waiting times, leading to increased response times and reduced efficiency.

5.No Consideration for I/O-bound Processes: SJF primarily focuses on CPU burst time, disregarding I/O-bound processes. If a process frequently requires I/O operations, SJF may not effectively prioritize such processes, leading to suboptimal performance in certain scenarios.

Round Robin

Introduction:

Round Robin (RR) is a CPU scheduling algorithm designed to provide fair allocation of CPU time among processes in a multitasking environment. It operates on the principle of time-sharing, where each process is assigned a fixed time slice or quantum to execute on the CPU before being preempted to allow another process to run. RR is widely used in operating systems and networking systems for its simplicity and ability to prevent any single process from monopolizing the CPU for an extended period. Let me know if you need more details on Round Robin!

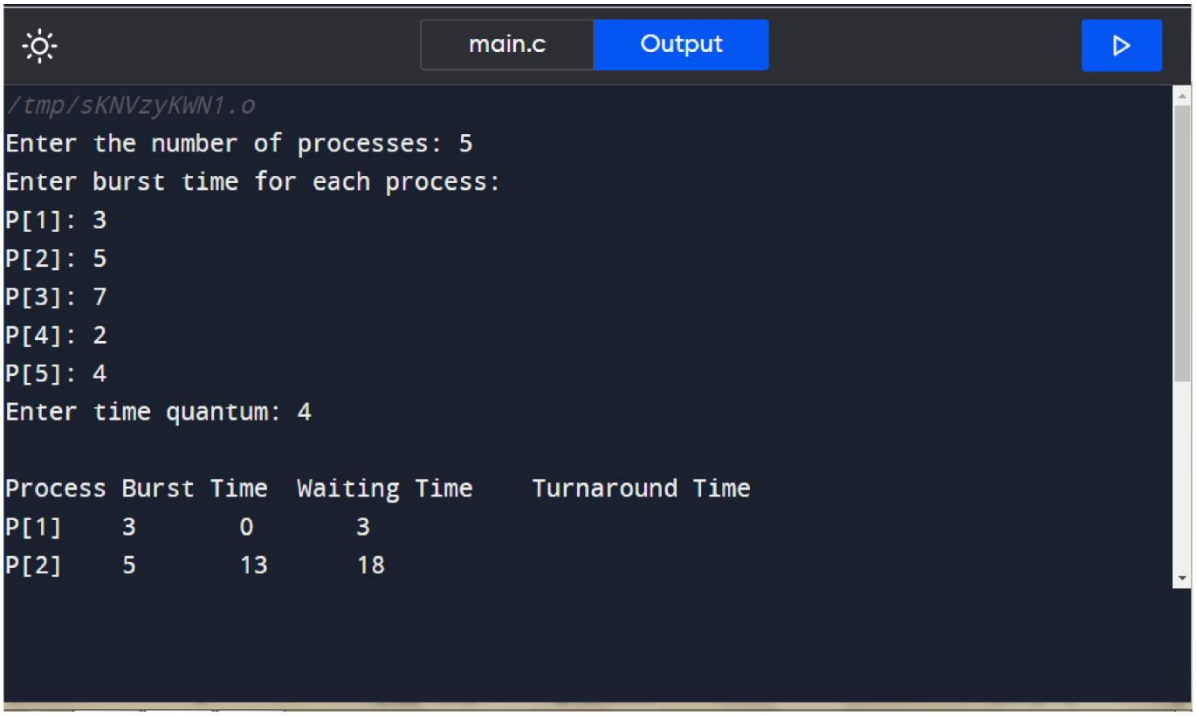
Code:

```
#include<stdio.h>


int main() {
    int n, quantum, i, j;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burst_time[n], remaining_time[n], waiting_time[n], turnaround_time[n];
    printf("Enter burst time for each process:\n");
```

```
for(i = 0; i < n; i++) {
    printf("P[%d]: ", i+1);
    scanf("%d", &burst_time[i]);
    remaining_time[i] = burst_time[i];
}
printf("Enter time quantum: ");
scanf("%d", &quantum);
int time = 0;
while(1) {
    int done = 1;
    for(i = 0; i < n; i++) {
        if(remaining_time[i] > 0) {
            done = 0;
            if(remaining_time[i] > quantum) {
                time += quantum;
                remaining_time[i] -= quantum;
            }
            else {
                time += remaining_time[i];
                waiting_time[i] = time - burst_time[i];
                remaining_time[i] = 0;
            }
        }
    }
    if(done == 1)
        break;
}
```

```
    for(i = 0; i < n; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
    }
    float avg_waiting_time = 0, avg_turnaround_time = 0;
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]\t%d\t\t%d\t\t%d\n", i+1, burst_time[i], waiting_time[i],
turnaround_time[i]);
        avg_waiting_time += waiting_time[i];
        avg_turnaround_time += turnaround_time[i];
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;
    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    return 0;
}
```

Input & Output:



Advantages:

Here are five advantages of the Round Robin (RR) scheduling algorithm:

1.Fairness: RR provides fair CPU time allocation to all processes in the system. Each process is given an equal share of CPU time through predefined time slices, ensuring that no process is starved of resources for an extended period.

2.Simple Implementation: RR is relatively simple to implement compared to other scheduling algorithms, such as priority-based or multilevel feedback queue scheduling. Its straightforward design makes it suitable for real-time systems and environments where simplicity is preferred.

3.Response Time: RR offers good response times for interactive processes. Since each process gets a turn to execute within a short time slice, users experience prompt responses from the system, enhancing overall user satisfaction.

4.Low Turnaround Time: RR can achieve low turnaround times for processes, especially when the quantum size is appropriately chosen. By allowing processes to execute in a round-robin fashion, it ensures that processes complete their execution within a reasonable time frame.

5.Support for Time-sharing Systems: RR is well-suited for time-sharing systems where multiple users interact with the system concurrently. By providing each user or process with regular CPU time slices, RR enables smooth multitasking and efficient utilization of system resources.

Disadvantages:

Here are five disadvantages of the Round Robin (RR) scheduling algorithm:

1.Inefficient for Long-Running Processes: RR can be inefficient for long-running processes, as they are repeatedly preempted after each time slice, resulting in frequent context switches. This overhead can decrease overall system performance and increase context switch overhead.

2.Poor Performance with Varying Quantum Sizes: The performance of RR can vary significantly depending on the chosen quantum size. If the quantum size is too small, there may be frequent context switches, leading to increased overhead. Conversely, if the quantum size is too large, shorter processes may experience increased waiting times.

3.Unpredictable Response Time: In RR, the response time for processes can be unpredictable, especially when the system is heavily loaded. Processes with shorter burst times may still need to wait for their turn, leading to variable response times and potentially impacting user experience.

4.Potential Starvation of I/O-Bound Processes: RR may not effectively handle I/O-bound processes, which spend a significant portion of their time waiting for I/O operations to complete. These processes may repeatedly relinquish the CPU voluntarily, resulting in inefficient CPU utilization and potential starvation.

5.Limited Performance Improvement for CPU-Bound Workloads: While RR can provide fair CPU time allocation, it may not significantly improve performance for CPU-bound workloads compared to other scheduling algorithms, such as Shortest Job First (SJF) or priority scheduling. This is because RR does not prioritize processes based on their execution time or importance.