

**Lab No: 10**

**Date: 2081/12/21**

**Write a program in C to find the factorial of a given number.**

---

**Theory:**

Factorial is the product of all positive integers from 1 to n. It is denoted as **n!** and is mathematically defined as:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1$$

$$n! = n \times (n-1)!$$

**Algorithm:**

1. Start
2. Define a function factorial(n):
  - If  $n \leq 1$ , return 1 (base case).
  - Otherwise, return  $n * \text{factorial}(n-1)$ .
3. In the main() function:
  - Declare an integer n.
  - Prompt the user to enter a number.
  - Read the input value n.
  - Call the factorial(n) function and print the result.
4. End

**Source Code:**

```
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) return 1;
    else return n * factorial(n - 1);
}

int main() {
    int n;

    printf("Enter a number: ");
```

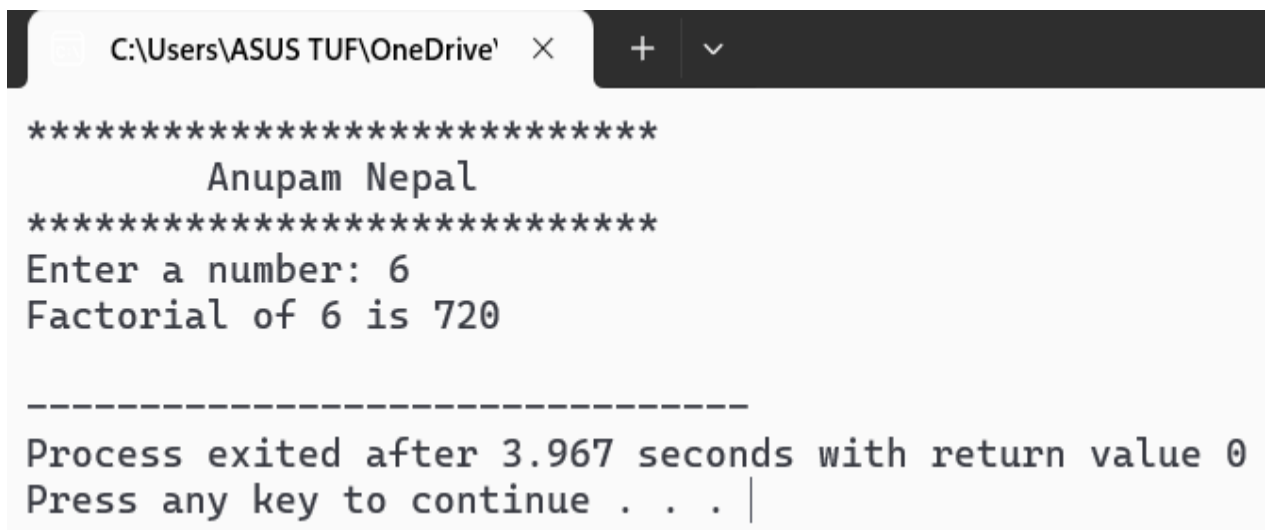
```
scanf("%d", &n);

printf("Factorial of %d is %d\n", n, factorial(n));

return 0;

}
```

### OUTPUT:



```
C:\Users\ASUS TUF\OneDrive' X + v

*****
      Anupam Nepal
*****
Enter a number: 6
Factorial of 6 is 720

-----
Process exited after 3.967 seconds with return value 0
Press any key to continue . . . |
```

**Lab No: 11**

**Date: 2081/12/21**

**Write a program in C to search an element in given array using sequential search.**

---

**Theory:**

Sequential search, also known as linear search, is a simple searching algorithm used to find an element in a list or array. It works by checking each element in the list one by one until the desired element is found or the entire list has been searched.

**Time Complexity:  $O(n)$**

**Algorithm:**

1. Start
2. Input an array A of n elements and a key x to search.
3. Set  $i=0$  (initialize index to start of the array).
4. While  $i < n$  do:
  - If  $A[i] == x$ , return i(element found at index i).
  - Otherwise, move to the next index ( $i = i+1$ ).
5. If the loop completes and x is not found, return -1.
6. End

**Source Code:**

```
#include <stdio.h>

int linearSearch(int arr[], int n, int target) {
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

```
int main() {  
  
    int n, target,i;  
  
    printf("Enter the number of elements: ");  
  
    scanf("%d", &n);  
  
    int arr[n];  
  
  
    printf("Enter the elements: ");  
  
    for (i = 0; i < n; i++) {  
  
        scanf("%d", &arr[i]);  
  
    }  
  
  
    printf("Enter the target value to search: ");  
  
    scanf("%d", &target);  
  
  
    int result = linearSearch(arr, n, target);  
  
    if (result == -1) {  
  
        printf("Target %d not found in the array.\n", target);  
  
    } else {  
  
        printf("Target %d found at index: %d\n", target, result);  
  
    }  
  
    return 0;  
}
```

## OUTPUT:

```
C:\Users\ASUS TUF\OneDrive' X + v
*****
          ANUPAM Nepal
*****
Enter the number of elements: 9
Enter the elements: 1 3 5 7 8 10 13 15 17
Enter the target value to search: 13
Target 13 found at index: 6

-----
Process exited after 35.4 seconds with return value 0
Press any key to continue . . . |
```

**Lab No: 12**

**Date: 2081/12/21**

**Write a program in C to sort an array using Heap Sort.**

---

**Theory:**

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It divides its input into a sorted and an unsorted region and iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.

**Time Complexity:  $O(n \log n)$**

**Source Code:**

```
#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 arr[] = { 12, 10, 23, 5, 2, 7 };

    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: \n");

    printArray(arr, n);

    heapSort(arr, n);

    printf("Sorted array using Heap Sort: \n");

    printArray(arr, n);

    return 0;

}

```

## OUTPUT:

```
C:\Users\ASUS TUF\OneDrive' X + v
*****
      Anupam Nepal
*****
Original array:
12 10 23 5 2 7
Sorted array using Heap Sort:
2 5 7 10 12 23

-----
Process exited after 0.08041 seconds with return value 0
Press any key to continue . . . |
```



**Lab No: 13**

**Date: 2081/12/21**

**Write a program in C to solve the Fractional Knapsack problem.**

---

**Theory:**

The fractional knapsack problem is a combinatorial optimization problem where the goal is to fill a container (the "knapsack") with fractional amounts of different materials to maximize the value of the selected materials. It uses a greedy algorithm by selecting items based on the highest value-to-weight ratio first.

**Time Complexity:  $O(n)$**

**Algorithm:**

1. Start
2. Calculate Value-to-Weight Ratios.
4. Sort Items by Ratio.
5. Fill the Knapsack.
  - Initialize total\_value = 0 and current\_weight = 0
  - For each sorted item:
    - If the full item fits:
      - Add entire item to knapsack
      - Update total\_value and current\_weight
    - Else:
      - Take the possible fraction of the item
      - Add partial value to total\_value
      - Break (knapsack is full)
6. Return total\_value as maximum achievable value.
7. End.

**Source Code:**

```
#include <stdio.h>

#include <stdlib.h>

struct Item {

    int value;

    int weight;
```

```

    double ratio;

};

int compare(const void *a, const void *b) {

    struct Item *itemA = (struct Item *)a;

    struct Item *itemB = (struct Item *)b;

    if (itemB->ratio > itemA->ratio) return 1;

    else if (itemB->ratio < itemA->ratio) return -1;

    else return 0;

}

double fractionalKnapsack(struct Item arr[], int n, int capacity) {

    int step = 1,i;

    double total_value = 0.0;

    int current_weight = 0;

    printf("Step %d: Computing value-to-weight ratios\n", step);

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

        arr[i].ratio = (double)arr[i].value / arr[i].weight;

        printf("Item %d: Value = %d, Weight = %d, Ratio = %.2f\n", i, arr[i].value, arr[i].weight,
arr[i].ratio);

    }

    step++;

    printf("Step %d: Sorting items by ratio (descending)\n", step);

    qsort(arr, n, sizeof(struct Item), compare);

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

        printf("Item %d: Value = %d, Weight = %d, Ratio = %.2f\n", i, arr[i].value, arr[i].weight,
arr[i].ratio);

    }

    step++;

    printf("Step %d: Filling knapsack (capacity = %d)\n", step, capacity);

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

        printf("Considering item %d: Value = %d, Weight = %d, Ratio = %.2f\n",

```

```

        i, arr[i].value, arr[i].weight, arr[i].ratio);

    printf("Current weight = %d, Remaining capacity = %d\n", current_weight, capacity -
current_weight);

    if (current_weight + arr[i].weight <= capacity) {

        current_weight += arr[i].weight;

        total_value += arr[i].value;

        printf("Fully adding item: Value = %d, Weight = %d\n", arr[i].value, arr[i].weight);

    } else {

        double fraction = (double)(capacity - current_weight) / arr[i].weight;

        total_value += arr[i].value * fraction;

        printf("Adding fraction %.2f of item: Value = %.2f, Weight = %.2f\n",

            fraction, arr[i].value * fraction, arr[i].weight * fraction);

        current_weight = capacity;

        break;

    }

    printf("Knapsack after step %d: Total Value = %.2f, Total Weight = %d\n", step, total_value,
current_weight);

    step++;

}

return total_value;

}

int main() {

    int n, capacity, i;

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

    scanf("%d", &n);

    struct Item arr[n];

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

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

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

```

```

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

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

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

}

printf("Enter the knapsack capacity: ");

scanf("%d", &capacity);

double max_value = fractionalKnapsack(arr, n, capacity);

printf("Maximum value achievable: %.2f\n", max_value);

return 0;

}

```

### OUTPUT:

```

C:\Users\ASUS TUF\OneDrive' X + v

Item 1 - Value: 100
Item 1 - Weight: 20
Item 2 - Value: 120
Item 2 - Weight: 30
Enter the knapsack capacity: 50
Step 1: Computing value-to-weight ratios
Item 0: Value = 60, Weight = 10, Ratio = 6.00
Item 1: Value = 100, Weight = 20, Ratio = 5.00
Item 2: Value = 120, Weight = 30, Ratio = 4.00
Step 2: Sorting items by ratio (descending)
Item 0: Value = 60, Weight = 10, Ratio = 6.00
Item 1: Value = 100, Weight = 20, Ratio = 5.00
Item 2: Value = 120, Weight = 30, Ratio = 4.00
Step 3: Filling knapsack (capacity = 50)
Considering item 0: Value = 60, Weight = 10, Ratio = 6.00
Current weight = 0, Remaining capacity = 50
Fully adding item: Value = 60, Weight = 10
Knapsack after step 3: Total Value = 60.00, Total Weight = 10
Considering item 1: Value = 100, Weight = 20, Ratio = 5.00
Current weight = 10, Remaining capacity = 40
Fully adding item: Value = 100, Weight = 20
Knapsack after step 4: Total Value = 160.00, Total Weight = 30
Considering item 2: Value = 120, Weight = 30, Ratio = 4.00
Current weight = 30, Remaining capacity = 20
Adding fraction 0.67 of item: Value = 80.00, Weight = 20.00
Maximum value achievable: 240.00

-----
Process exited after 35.44 seconds with return value 0
Press any key to continue . . . |

```

**Lab No: 14**

**Date: 2081/12/21**

**Write a program in C to demonstrate Job Sequencing with deadlines.**

---

**Theory:**

The Job Sequencing Problem is a classic greedy algorithm where each job takes one unit of time, has a deadline (i.e., the latest time it must be completed). The goal is to schedule the jobs such that the maximum total profit is earned. You can only schedule one job per time unit, and no job can be done after its deadline.

**Algorithm:**

1. Start
2. Sort Jobs by Profit.
3. Find Maximum Deadline.
4. Schedule Jobs in Slots
  - For each job (in order of sorted list):
    - Find the latest available time slot  $\leq$  job's deadline
    - If a slot is free, assign the job to that slot.
    - If not, skip the job (it can't be scheduled).
  - Update the total profit after each assignment.
5. Output Final Result
  - Print the job IDs in the order they were scheduled.
  - Print the maximum total profit earned.
6. End

**Source Code:**

```
#include <stdio.h>

#include <stdlib.h>

struct Job {

    int id, profit, deadline;

};

int compare(const void *a, const void *b) {

    struct Job *jobA = (struct Job *)a;

    struct Job *jobB = (struct Job *)b;
```

```

    return jobB->profit - jobA->profit;
}

void jobSequencing(struct Job arr[], int n) {

    int step = 1, i, j, k, total_profit = 0;

    printf("Step %d: Sorting jobs by profit (descending)\n", step);

    qsort(arr, n, sizeof(struct Job), compare);

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

        printf("Job %d: Profit = %d, Deadline = %d\n", arr[i].id, arr[i].profit, arr[i].deadline);

    }

    step++;

    int max_deadline = 0;

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

        if (arr[i].deadline > max_deadline)

            max_deadline = arr[i].deadline;

    }

    printf("Step %d: Maximum deadline = %d\n", step, max_deadline);

    step++;

    int slots[max_deadline];

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

        slots[i] = -1;

    printf("Step %d: Scheduling jobs\n", step);

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

        printf("Considering Job %d: Profit = %d, Deadline = %d\n",

            arr[i].id, arr[i].profit, arr[i].deadline);

        for (j = (arr[i].deadline < max_deadline ? arr[i].deadline : max_deadline) - 1; j >= 0; j--) {

            printf("Checking slot %d: ", j);

            if (slots[j] == -1) {

                slots[j] = arr[i].id;

                total_profit += arr[i].profit;
            }
        }
    }
}

```

```

        printf("Assigned to Job %d\n", arr[i].id);

        printf("Schedule after step %d: ", step);

        for (k = 0; k < max_deadline; k++) {

            if (slots[k] == -1) printf("_ ");

            else printf("J%d ", slots[k]);

        }

        printf("(Profit = %d)\n", total_profit);

        step++;

        break;

    } else {

        printf("Occupied by Job %d\n", slots[j]);

    }

}

printf("Final Schedule: ");

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

    if (slots[i] != -1)

        printf("J%d ", slots[i]);

}

printf("\nMaximum profit achievable: %d\n", total_profit);

}

int main() {

    int n, i;

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

    scanf("%d", &n);

    struct Job arr[n];

    printf("Enter job ID, profit, and deadline for each job:\n");

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

        printf("Job %d - ID: ", i);

```

```

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

printf("Job %d - Profit: ", i);

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

printf("Job %d - Deadline: ", i);

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

}

jobSequencing(arr, n);

return 0;

}

```

## OUTPUT:

```

C:\Users\ASUS TUF\OneDrive' X + v
*****
          Anupam Nepal
*****
Enter the number of jobs: 3
Enter job ID, profit, and deadline for each job:
Job 0 - ID: 1
Job 0 - Profit: 100
Job 0 - Deadline: 2
Job 1 - ID: 2
Job 1 - Profit: 27
Job 1 - Deadline: 2
Job 2 - ID: 3
Job 2 - Profit: 25
Job 2 - Deadline: 1
Step 1: Sorting jobs by profit (descending)
Job 1: Profit = 100, Deadline = 2
Job 2: Profit = 27, Deadline = 2
Job 3: Profit = 25, Deadline = 1
Step 2: Maximum deadline = 2
Step 3: Scheduling jobs
Considering Job 1: Profit = 100, Deadline = 2
Checking slot 1: Assigned to Job 1
Schedule after step 3: _ J1 (Profit = 100)
Considering Job 2: Profit = 27, Deadline = 2
Checking slot 1: Occupied by Job 1
Checking slot 0: Assigned to Job 2
Schedule after step 4: J2 J1 (Profit = 127)
Considering Job 3: Profit = 25, Deadline = 1
Checking slot 0: Occupied by Job 2
Final Schedule: J2 J1
Maximum profit achievable: 127

-----
Process exited after 33.92 seconds with return value 0
Press any key to continue . . . |

```