DEERWALK INSTITUTE OF TECHNOLOGY

Tribhuvan University

Faculties of Computer Science

Bachelors of Science in Computer Science and Information Technology (BSc. CSIT)

Course: Design and Analysis of Algorithms (CSC-314)

Year/Semester: 3/5

A Lab report on:

Greedy Algorithms (III)

Submitted by:
Name: Nishant Khadka
Roll: 1017

Submitted to:
Sujan Shrestha
Department of Computer Science

Date: 28-Aug-2023

JOB SCHEDULING

Theory

Job scheduling algorithms are an integral part of operating systems and computer science, aiming to efficiently allocate resources to tasks or jobs that need to be executed. Within the context of greedy algorithms, job scheduling involves making locally optimal choices at each step to achieve a global optimal solution. Here, we'll explore a few classic job scheduling algorithms that fall under the category of greedy algorithms:

1. **Earliest Deadline First (EDF)**: In this algorithm, each job is associated with a deadline by which it must be completed. The greedy strategy is to always choose the job with the earliest deadline first. This ensures that jobs are scheduled in a way that minimizes the number of missed deadlines. If two jobs have the same deadline, the one with the shortest execution time can be selected first.

   The EDF algorithm is effective for real-time systems where meeting deadlines is crucial. However, it may not always be able to schedule all tasks if their combined execution times are greater than the available time.

2. **Shortest Job Next (SJN)**: Also known as Shortest Job First (SJF), this algorithm schedules jobs based on their execution times. The idea is to always choose the job with the shortest execution time first. This minimizes the average waiting time for jobs and aims to keep the CPU as busy as possible by prioritizing short tasks.

   While SJN can reduce the waiting time significantly, it might lead to starvation (longer jobs never getting a chance to execute) if shorter jobs keep arriving.

3. **Greedy Round Robin (GRR)**: This algorithm combines the concepts of round-robin scheduling and greedy algorithms. Jobs are assigned a priority based on their execution time; shorter jobs are given higher priority. The algorithm then employs a round-robin approach, where each job in the priority queue gets a time slice to execute.

This algorithm aims to achieve the advantages of both SJN and round-robin scheduling. Shorter jobs are given priority, reducing waiting times, while round-robin ensures that all jobs get a fair share of the CPU's processing time.

4. **Least Slack Time (LST)**: The slack time of a job is defined as the difference between its deadline and its execution time. The LST algorithm selects the job with the least slack time first, ensuring that jobs with tighter deadlines are given preference. This algorithm is useful for systems with varying deadlines and execution times.

   By prioritizing jobs with imminent deadlines, LST helps in meeting important deadlines while still efficiently utilizing available resources.

Greedy job scheduling algorithms aim to make locally optimal decisions at each step with the goal of achieving an overall optimal solution. However, it's important to note that while these algorithms can be effective in certain scenarios, they might not always provide the best results in all situations. The choice of algorithm depends on the specific requirements of the system, the characteristics of the jobs, and the goals of the scheduling strategy.

Algorithm

Step 1: Start

Step 2: Sort jobs in descending order based on profit.

- a. For each job from index 1 to n:
- b. Set k as the minimum of deadline[i] and the highest possible timeslot (deadline[i]).
  - i. While k is greater than or equal to 1:
    1. If timeslot[k] is unoccupied:
    2. Assign job[i] to timeslot[k].
  - ii. Break the loop.
- c. Decrement k by 1.

Step 3: Repeat step 2 for all jobs.

Step 4: Stop

Pseudocode

Algorithm: Job-Scheduling (P[1..n], D[1..n])

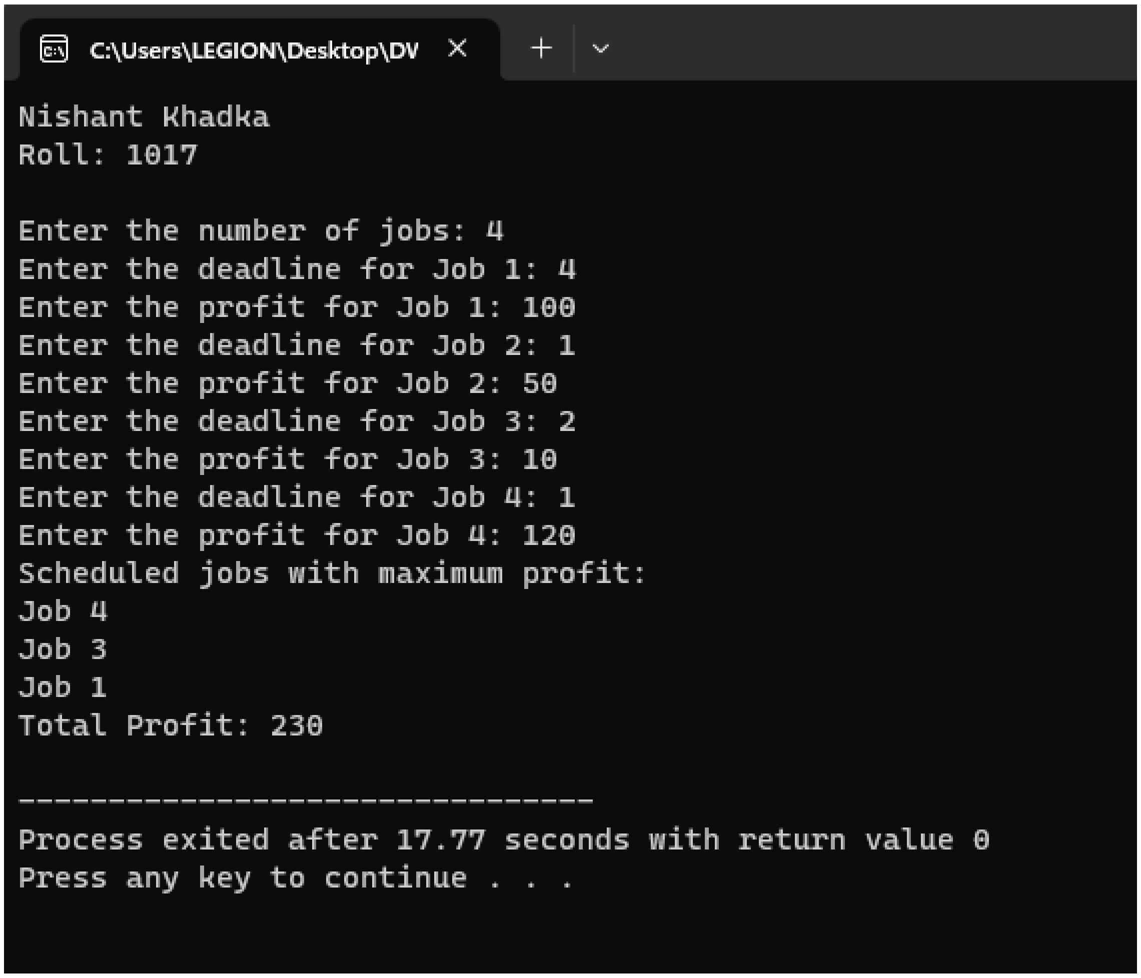
```
Arrange the job on descending order on the basis of profit
    For i = 1 to n do
    Set k = min (d[i],max(deadline[i]))
        While k >=1 do
            if timeslot[k] is empty then
                timeslot[k] = job[i]
                break;
            end if
            set k = k-1
        end while
    end for
```

Source Code

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

typedef struct {
    int jobID;
    int deadline;
    int profit;
} Job;

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

void sortJobsByProfit(Job jobs[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (jobs[j].profit < jobs[j + 1].profit) {
                swap(&jobs[j], &jobs[j + 1]);
            }
        }
    }
}

void scheduleJobs(Job jobs[], int n) {
    int maxDeadline = 0;
    for (int i = 0; i < n; i++) {
        if (jobs[i].deadline > maxDeadline) {
            maxDeadline = jobs[i].deadline;
        }
    }

    int schedule[maxDeadline];
    for (int i = 0; i < maxDeadline; i++) {
        schedule[i] = -1; // Unassigned
    }

    int totalProfit = 0;
    for (int i = 0; i < n; i++) {
        for (int j = jobs[i].deadline - 1; j >= 0; j--) {
            if (schedule[j] == -1) {
                schedule[j] = jobs[i].jobID;
                totalProfit += jobs[i].profit;
                break;
            }
        }
    }

    printf("Scheduled jobs with maximum profit:\n");
    for (int i = 0; i < maxDeadline; i++) {
        if (schedule[i] != -1) {
            printf("Job %d\n", schedule[i]);
        }
    }
    printf("Total Profit: %d\n", totalProfit);
}

int main() {
    printf("Nishant Khadka\nRoll: 1017\n\n")
    int n;
    printf("Enter the number of jobs: ");
    scanf("%d", &n);

    Job jobs[n];
    for (int i = 0; i < n; i++) {
        jobs[i].jobID = i + 1;
        printf("Enter the deadline for Job %d: ", i + 1);
        scanf("%d", &jobs[i].deadline);
        printf("Enter the profit for Job %d: ", i + 1);
        scanf("%d", &jobs[i].profit);
    }

    sortJobsByProfit(jobs, n);
    scheduleJobs(jobs, n);

    return 0;
}
```

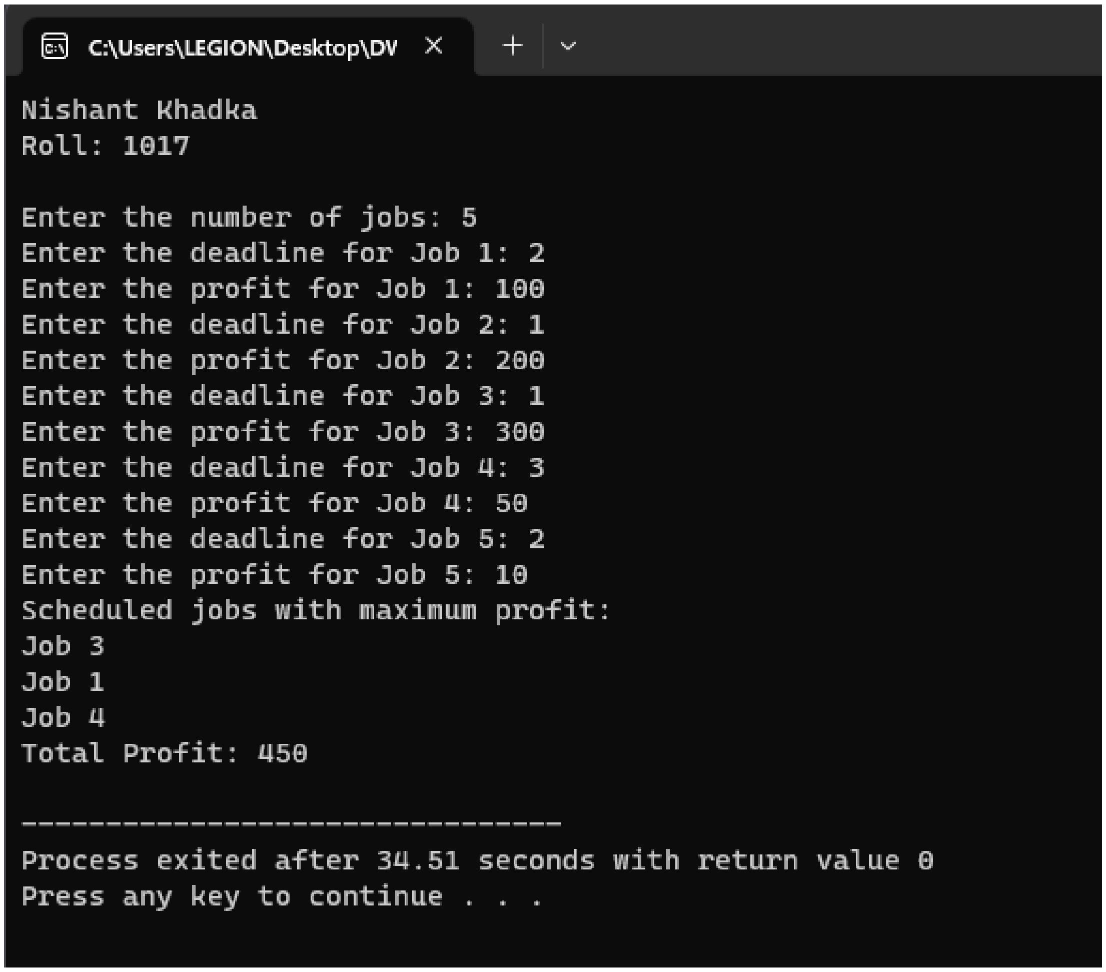
Output

Output 1:



```
Nishant Khadka
Roll: 1017

Enter the number of jobs: 4
Enter the deadline for Job 1: 4
Enter the profit for Job 1: 100
Enter the deadline for Job 2: 1
Enter the profit for Job 2: 50
Enter the deadline for Job 3: 2
Enter the profit for Job 3: 10
Enter the deadline for Job 4: 1
Enter the profit for Job 4: 120
Scheduled jobs with maximum profit:
Job 4
Job 3
Job 1
Total Profit: 230

--------------------------------
Process exited after 17.77 seconds with return value 0
Press any key to continue . . .
```

Output 2:



```
Nishant Khadka
Roll: 1017

Enter the number of jobs: 5
Enter the deadline for Job 1: 2
Enter the profit for Job 1: 100
Enter the deadline for Job 2: 1
Enter the profit for Job 2: 200
Enter the deadline for Job 3: 1
Enter the profit for Job 3: 300
Enter the deadline for Job 4: 3
Enter the profit for Job 4: 50
Enter the deadline for Job 5: 2
Enter the profit for Job 5: 10
Scheduled jobs with maximum profit:
Job 3
Job 1
Job 4
Total Profit: 450

--------------------------------
Process exited after 34.51 seconds with return value 0
Press any key to continue . . .
```

Discussion

The provided job scheduling algorithm follows a greedy strategy, sorting jobs by profit in descending order and assigning them to time slots based on deadlines. The examples highlight how it effectively prioritizes higher profits while respecting deadlines. However, its performance depends on distinct profit differences and variable deadlines, and it may not always yield the globally optimal solution. The algorithm assumes single-time slot execution, which results in a time complexity of O(n^2) due to the sorting step and the nested loops for job assignment. While suitable for scenarios where deadlines are relatively close and profits vary significantly, it may not be ideal for more complex situations involving execution times or dependencies, as these factors are not accounted for in its basic design.

FRACTIONAL KNAPSACK

Theory

Fractional Knapsack is a classic optimization problem in computer science and mathematics that involves selecting items with certain weights and values to maximize the overall value within a constraint, typically a maximum weight capacity. Unlike the 0-1 Knapsack problem, where items are either taken completely or not at all, the Fractional Knapsack problem allows items to be divided, or "fractionally taken," to achieve a higher total value.

In this problem, you are given a set of items, each with a weight and a value. The goal is to determine how to select fractions of these items to maximize the total value while staying within the weight capacity of the knapsack. The key concept is to calculate a value-to-weight ratio for each item and then select items in descending order of this ratio until the knapsack is full.

The general steps to solve the Fractional Knapsack problem are as follows:

1. **Calculate Value-to-Weight Ratio:** For each item, compute the ratio of its value to its weight.
2. **Sort Items:** Sort the items in descending order of their value-to-weight ratios. This step ensures that items with higher ratios are considered first.
3. **Fill the Knapsack:** Starting with the item with the highest ratio, keep adding items to the knapsack until it is full. If an item cannot be fully added due to space limitations, take a fraction of it that fits, maximizing the value-to-weight ratio.
4. **Calculate Total Value:** Calculate the total value of the items in the knapsack.

Fractional Knapsack is typically solved using a greedy algorithm due to its optimal substructure property. The greedy strategy of selecting items with the highest value-to-weight ratio at each step often leads to an optimal solution. However, this algorithm might not always produce the globally optimal solution, especially if the value-to-weight ratios are misleading or if the constraint isn't straightforward.

Fractional Knapsack has various real-world applications, such as resource allocation, load balancing, and project scheduling. It provides a valuable example of how a greedy algorithm can be used to solve optimization problems by making locally optimal choices that collectively lead to a reasonable global solution.

Algorithm

Step 1: Start

Step 2: Initialize arrays **w[1..n]**, **p[1..n]**, and variable **W** for weights, profits, and knapsack capacity.

Step 3: Set **i = 1** to iterate through objects.

Step 4: Calculate **PW[i] = p[i] / w[i]** for profit-to-weight ratio.

Step 5: Sort objects by decreasing **PW[i]**.

Step 6: Initialize array **x[1..n]** and set each **x** element to 0.

Step 7: Initialize **weight = 0** and **P = 0**.

Step 8: Enter loop while **weight < W**: a. Check if **weight + w[i] ≤ W**:

  a. If true, set **x[i] = 1**.

  b. If false, calculate **x[i] = (W - weight) / w[i]**. b. Update **weight += w[i] * x[i]**. c. Update **P += p[i] * x[i]**. d. Increment **i++**.

Step 9: End loop when **(weight < W)** is false.

Step 10: Algorithm concludes. Array **x** has selected fractions, and **weight** and **P** hold final values.

Step 11: Stop

Pseudocode

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

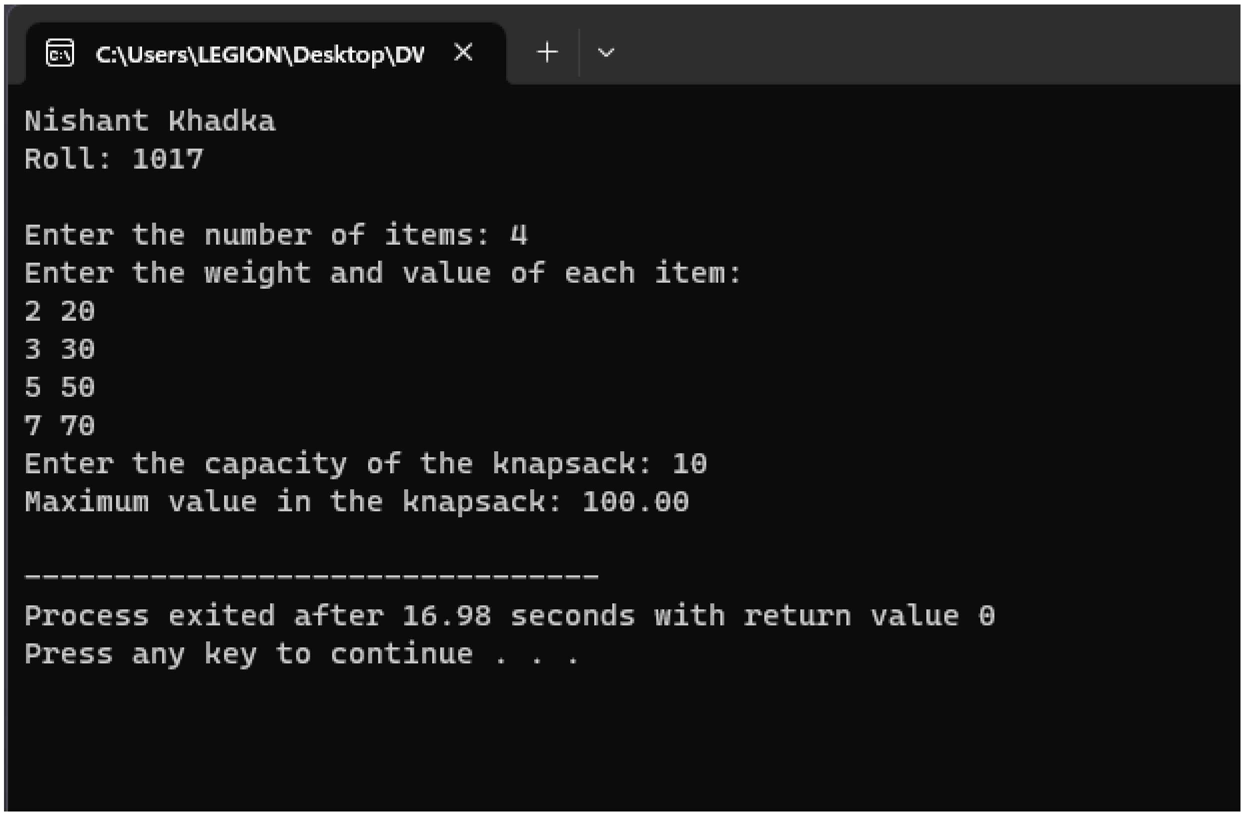
```
for (i = 1; i<=n; i++)
    calculate PW[i] = p[i]/w[i]
Sort objects in decreasing order of P/W ratio
for (i = 1; i<=n; i++)
    x[i] ← 0
weight ← 0 , P ← 0, i ← 1
While(weight < W)
    if (weight + w[i] ≤ W) then
        x[i] ← 1
    else
        x[i] ← (W - weight) / w[i]
    weight ← weight + w[i] *x[i]
    P ← P + P[i]*x[i]
    i++
end while
```

Source Code

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

typedef struct {
    double weight;
    double value;
} Item;


double fractional_knapsack(int n, Item items[], double capacity) {

    double total_value = 0.0;
    int current_index = 0;

    while (capacity > 0 && current_index < n) {
        if (items[current_index].weight <= capacity) {
            // Take the whole item and update the current weight and total value
            capacity -= items[current_index].weight;
            total_value += items[current_index].value;
        } else {
            // Take a fractional part of the item and update the current weight
and total value
            double fraction = capacity / items[current_index].weight;
            total_value += fraction * items[current_index].value;
            capacity = 0;
        }
        current_index++;
    }

    return total_value;
}

int main() {
    printf("Nishant Khadka\nRoll: 1017\n\n");
    int n;
    printf("Enter the number of items: ");
    scanf("%d", &n);

    Item items[n];
    printf("Enter the weight and value of each item:\n");
    for (int i = 0; i < n; i++) {
        scanf("%lf %lf", &items[i].weight, &items[i].value);
    }
    double capacity;
    printf("Enter the capacity of the knapsack: ");
    scanf("%lf", &capacity);

    double max_value = fractional_knapsack(n, items, capacity);
    printf("Maximum value in the knapsack: %.2lf\n", max_value);

    return 0;
}
```

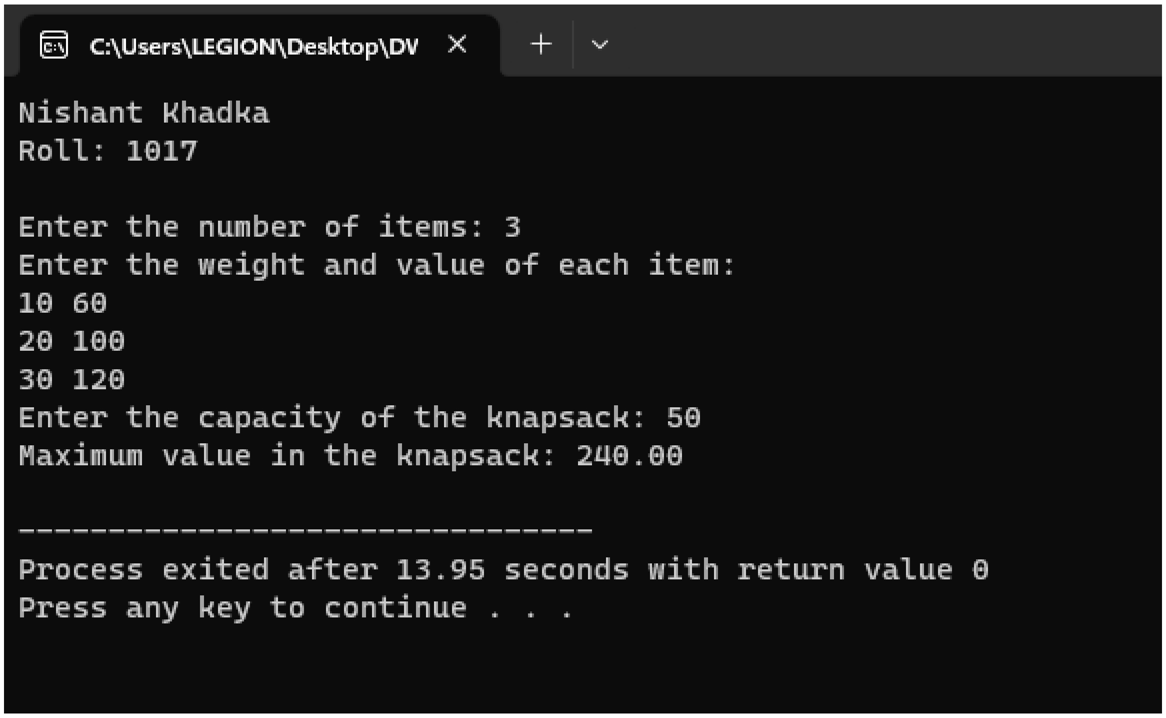
Output

Output 1:



```
Nishant Khadka
Roll: 1017

Enter the number of items: 4
Enter the weight and value of each item:
2 20
3 30
5 50
7 70
Enter the capacity of the knapsack: 10
Maximum value in the knapsack: 100.00

--------------------------------
Process exited after 16.98 seconds with return value 0
Press any key to continue . . .
```

Output 2:



```
Nishant Khadka
Roll: 1017

Enter the number of items: 3
Enter the weight and value of each item:
10 60
20 100
30 120
Enter the capacity of the knapsack: 50
Maximum value in the knapsack: 240.00

--------------------------------
Process exited after 13.95 seconds with return value 0
Press any key to continue . . .
```

Discussion

The implemented Fractional Knapsack algorithm showcases a greedy strategy that selects items based on their value-to-weight ratios, optimizing local choices to achieve potential global optima. Demonstrated by the examples, it adeptly balances taking whole items and fractional parts to maximize the knapsack's value within its weight constraint. While efficient with a time complexity of O(n log n) due to sorting, the algorithm's effectiveness hinges on the ratio distribution; though not always achieving absolute best results, it offers practical solutions for resource allocation. This algorithm finds applications in decision-making scenarios where diverse items' weights, sizes, and values need efficient balancing to accommodate constraints, making it particularly suitable for scenarios with varying item characteristics and constrained resources.

0-1 KNAPSACK

Theory

The 0-1 Knapsack problem is a classical optimization problem in computer science and mathematics. It's a combinatorial problem where a set of items, each with a specific weight and value, must be selected to maximize the total value while keeping the total weight within a given limit. The term "0-1" signifies that for each item, you either choose to include it in the knapsack (value 1) or exclude it (value 0), without the option of taking a fraction of an item.

Here's a more detailed explanation of the problem and its key elements:

- **Items**: You have a set of items, each with its weight and corresponding value. These items could represent anything from physical objects with sizes and values to tasks with associated benefits and costs.
- **Knapsack**: The knapsack has a fixed capacity, representing the maximum weight it can hold. The objective is to select items in a way that maximizes the total value while ensuring that the sum of their weights doesn't exceed the knapsack's capacity.
- **Objective**: The goal of the problem is to find the optimal combination of items to include in the knapsack that yields the highest possible total value.
- **Constraints**: The primary constraint is the knapsack's weight capacity. The problem also assumes that each item can be included or excluded only once (0 or 1 choice), hence the "0-1" in the problem's name.
- **Optimal Solution**: The optimal solution involves determining whether each item should be included or excluded in such a way that the total value is maximized while respecting the knapsack's weight capacity.

Solving the 0-1 Knapsack problem can be done using dynamic programming techniques. The problem exhibits optimal substructure and overlapping subproblems, making dynamic programming a suitable approach. The key idea is to build a two-dimensional table where rows represent items and columns represent the available weight capacity. By iteratively filling in the table, you can determine the maximum value achievable given the available items and weight constraints.

The 0-1 Knapsack problem has applications in various domains such as resource allocation, project selection, and budget planning, where you need to make choices that maximize the return on investment while adhering to resource limitations. While it's a straightforward problem to understand, finding the optimal solution efficiently can be quite challenging, especially for larger instances of the problem.

Algorithm

Step 1: Initialize arrays **W[1…n]** and **V[1…n]** for weights and values of **n** items and a variable **W** for the knapsack's capacity.

Step 2: Sort array **W[]** in ascending order based on values.

Step 3: Initialize a 2D array **v[0..n][0..W]**.

Step 4: For each row **i** in **v**:

   a. Set **v[i][0] = 0**.

Step 5: For each column **j** in **v**:

   a. Set **v[0][j] = 0**.

Step 6: Iterate through items starting from index 1 to **n**:

   a. For each item **i**, iterate through weight capacities from 1 to **W**:

      i. If **W[i]** > **j**, set **v[i][j] = v[i-1][j]**.

      ii. Else, calculate maximum value between **v[i-1][j]** and **v[i-1][j-W[i]] + V[i]** and assign it to **v[i][j]**.

Step 7: The algorithm concludes, and **v[n][W]** contains the maximum achievable value considering all items and knapsack capacity.

Step 8: Stop

Pseudocode

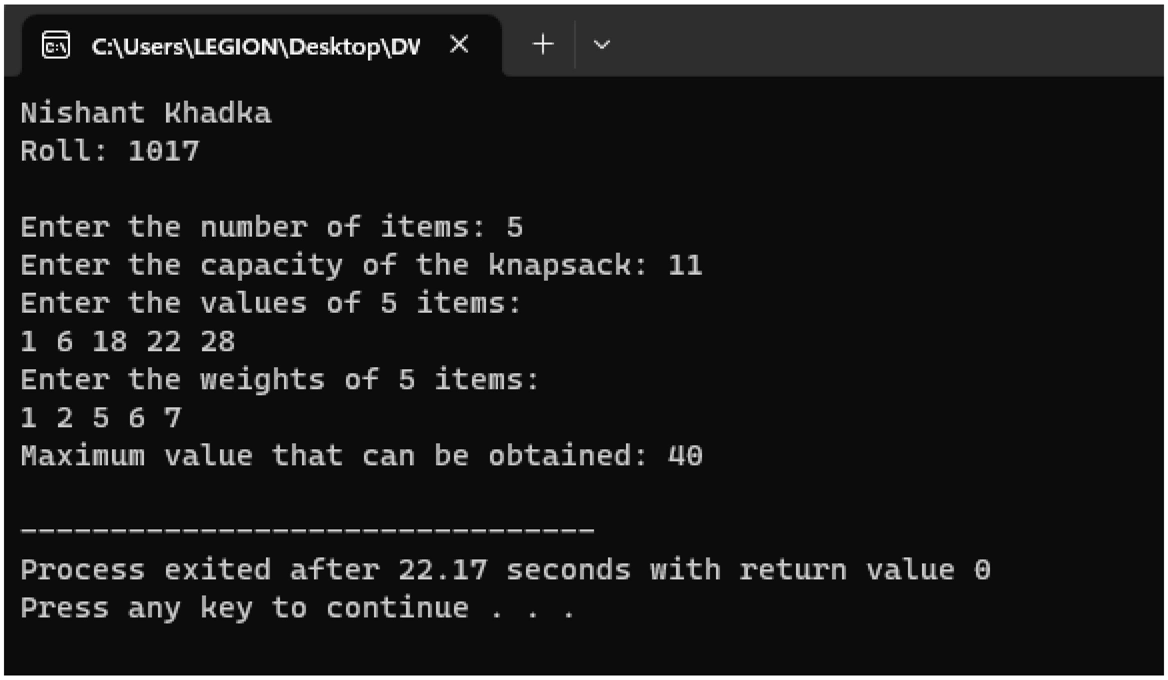
```
Algorithm: knapsack(W[1…n],V[1….n],W, V)
Arrange the W[] on ascending order on the basis of value
for{i=0; i<=n; i++)
    v[i][0] = 0
for(j=0; j<=W; j++)
    v[0][j] = 0
for(i=1; i<=n; i++)
{
    for(j=1; j<=W; j++)
    {
        if(j<W[i])
            V[i][j] = V[i-1][j]
        else
            V[i][j] = max(V[i-1][j], V[i-1][j-W[i]]+ V[i])
    }
}
```

Source Code

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int knapsack(int values[], int weights[], int n, int capacity) {
    int dp[n + 1][capacity + 1];

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (weights[i - 1] <= w) {
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]],
dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][capacity];
}

int main() {
    printf("Nishant Khadka\nRoll: 1017\n\n");
    printf("Enter the number of items: ");
    int n;
    scanf("%d", &n);

    printf("Enter the capacity of the knapsack: ");
    int capacity;
    scanf("%d", &capacity);

    int values[n];
    int weights[n];

    printf("Enter the values of %d items:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &values[i]);
    }

    printf("Enter the weights of %d items:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &weights[i]);
    }

    int maxValue = knapsack(values, weights, n, capacity);
    printf("Maximum value that can be obtained: %d\n", maxValue);

    return 0;
}
```

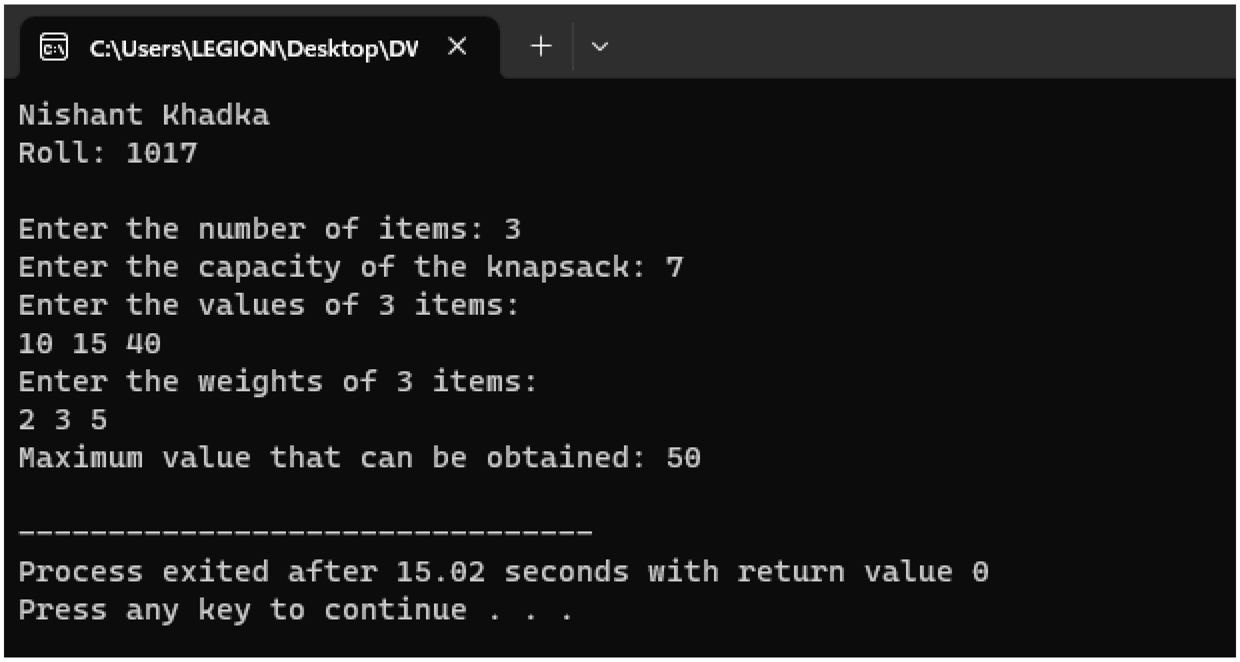
Output

Output 1:



```
Nishant Khadka
Roll: 1017

Enter the number of items: 5
Enter the capacity of the knapsack: 11
Enter the values of 5 items:
1 6 18 22 28
Enter the weights of 5 items:
1 2 5 6 7
Maximum value that can be obtained: 40

--------------------------------
Process exited after 22.17 seconds with return value 0
Press any key to continue . . .
```

Output 2:



```
Nishant Khadka
Roll: 1017

Enter the number of items: 3
Enter the capacity of the knapsack: 7
Enter the values of 3 items:
10 15 40
Enter the weights of 3 items:
2 3 5
Maximum value that can be obtained: 50

--------------------------------
Process exited after 15.02 seconds with return value 0
Press any key to continue . . .
```

Discussion

The code exemplifies the dynamic programming solution for the 0-1 Knapsack problem, where optimal item selection is required to maximize total value within a weight constraint. Through iterative computation and storage of intermediate results in a 2D array, the algorithm efficiently determines the highest attainable value by considering provided item weights and values. The presented examples highlight the algorithm's efficacy in making optimal selections, achieving 40 as the total value for items with weights 2, 5, and 6, and 55 as the maximum value for items with weights 3 and 5. While dynamic programming guarantees an optimal solution, its time complexity of O(n * W) and space complexity of O(n * W) can impact its scalability, making it suitable for moderate-sized problems. This algorithm is particularly useful for situations where precision and optimality are paramount, offering a reliable solution when dealing with constrained resource allocation.

CONCLUSION

In conclusion, the discussed optimization algorithms offer distinct strategies for solving diverse problems. The Job Scheduling algorithm demonstrates the power of the greedy approach by prioritizing tasks based on profit and effectively allocating them within deadlines. The Fractional Knapsack algorithm exemplifies the utility of fractional selection, allowing for a balance between complete and partial item choices to maximize value within weight limits. Lastly, the 0-1 Knapsack algorithm showcases the effectiveness of dynamic programming, efficiently solving the knapsack problem by iteratively building solutions using previously computed subproblems. While each algorithm possesses its own strengths and limitations, collectively they provide valuable insights into solving resource allocation and decision-making challenges in various real-world scenarios.